Parallelization Efficiency of Vectorized Codes:
iPSC/860 Case Studies

Robert J. Bergeron[1]

Report RND-93-016 November 1993

NAS Systems Development Branch
NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

Abstract

This paper describes the manual conversion of several vectorized Fortran algorithms to the iPSC/860. The codes typify NAS Y-MP solution techniques and represent the types of algorithms that automatic parallelizers must analyze and decompose. The straightforward, high-level domain decompositions employed for the explicit and SOR algorithms performed quite well. The implicit ADI algorithm required a new solver to reduce communication loads and a pipelined scheme to increase node utilization. The loop-level decomposition employed for the multigrid algorithm performed poorly. The major lesson taught by these manual transformations is that automatic tools must expose or utilize parallelism at a high level to create effective distributed memory codes. The postprocessor PAT utility assisted in disclosing reasons for less-than-expected parallel performance.

1.0 Introduction

[1] Computer Sciences Corporation, NASA Ames Research Center, Moffett Field, CA 94035-1000

The current limits on technology indicate that future high speed computer systems will employ multiple processors, operating in parallel, to solve computationally-intensive problems. Highly parallel machines have distributed memories and require coarse-grained partitioning to amortize the communication overhead incurred by transmitting messages over networks. Coarse-grained parallel execution requires substantial rewriting of singletasked source programs to enable independent execution of tasks larger than those performed by DO-loops. The past success of automatic vectorization has motivated the development of automatic parallelization tools to reduce the user effort in constructing parallel programs from single-tasked programs.

The parallel tools should help users construct code for task allocation, CPU synchronization, and interprocessor communication. Code generated by automatic tools for this purpose can assume many forms, depending upon the target parallel architecture and the transformation rules internal to the tool. Evaluation of coarse-grained tools needs at least one example of a source transformation for comparison against the tool-generated code. This report presents the results of user-written, higher-level transformations on these same sources for execution on a distributed memory machine. Comparison of the transformed source to those produced with the help of the parallel tool should help in the evaluation of automatic tools for highly parallel machines.

Typical steps in creating a parallel program consist of algorithm decomposition, mapping the calculations to the processors, and tuning the program to increase performance. Algorithm decomposition involves the separation of the calculation into a set of tasks which can act independently on distinct sets of data. Common strategies for segregating the calculations are based on geometry or control. Mapping these tasks to the processors consists in assigning parts of the calculation to the processors in such a way as to balance the computational load and minimize communication delays. Tuning the program involves those steps required to enhance performance. To increase the floating point performance of individual nodes, such tuning might involve replacement of critical sections of Fortran with assembler code or invoking specialized math libraries. Tuning might also require an improved mapping strategy or use of asynchronous message-passing to reduce the communication overhead.

Since the purpose of this work is to illustrate the sort of steps required to obtain efficient iPSC/860 code, this report will emphasize the first two areas.

1.1 The iPSC/860 Environment

All calculations reported here were performed on the NAS Intel iPSC/860, an Intel i860 processsor-based hypercube, with 128 processing nodes attached to an Intel 80386 host processor. The i860 node has an 8 Megabyte (MB) memory, an 8 Kilobyte (KB) cache and multiple arithmetic units which allow multiple operations per cycle. All calculations reported here were performed in 64-bit mode, making the data capacity of the i860 node memory effectively equal to 1 MW and the cache effectively equal to 1 KW. The i860 node has a peak performance of 40 MFLOPS in 64-bit mode.

Since the iPSC/860 is a Multiple Instruction, Multiple Data (MIMD) computer, individual nodes perform independently (asynchronously) unless coupled by synchronization. Moreover, the parallel processes which perform the calculations access data which is private to each process. Processes obtain needed data and achieve synchronization by exchanging messages with each other. The node hardware includes direct connect modules which relieve the node CPU of routing overhead. There is incentive for exchanging small messages. Messages of length less than 100 bytes are sent without a precursor message to verify the existence of sufficient data space on the receiving node. Messages exceeding 100 bytes require an additional round trip to verify the existence of the required space on the receiving node. Messages longer than 2000 bytes require additional system intervention as the 4000 byte FIFOs are emptied.

Intel also provides a Performance Analysis Tool (PAT) to help users measure a variety of parallel performance parameters (Intel, 1992). This utility gathers runtime performance data for later postprocessing. The tool has simple options for subroutine and message-passing timings and more complex options for tracing parallel program execution. Section 6 provides examples of PAT use and graphical output.

1.2 The Parallel Suite

This paper describes the modifications required to create iPSC/860 versions of five Fortran codes used previously to test the ability of the Cray Fortran PreProcessor (FPP) to generate efficient parallel code on the Cray Y-MP (Bergeron, 1993). The codes, employing numerical methods programmed for execution on a vector machine, typify solution techniques which NAS users would submit for parallelization by an automatic preprocessor. The suite includes the following kernels:

- Successive OverRelaxation (SOR) algorithm in a cube geometry,
- Shallow Water Model (SWM) providing an explicit solution to the two-dimensional mass and momentum equations treating wave motion,
- PARAllel Cyclic Reduction algorithm (PARACR) solving a two-dimensional tridiagonal system,
- A 2-D Alternating Direction Implicit (ADI) algorithm, and a
- MultiGRid (MGR) algorithm in a cube geometry.
- The following sections provide some iPSC/860 ground rules for evaluation and describe the performance of each of the five codes. The paper presents the codes in the order of difficulty. Implementation of the explicit codes, SOR and SWM, required much less effort than the implicit code. The implicit code required both a new solver and a pipelined implementation. The MGR employed the Y-MP

decomposition. The paper also presents some tool characteristics which would be desirable for parallelizing NAS workload programs.

2.0 Ground Rules and Observations

The following section provides ground rules and evaluation technique for parallel versions of the singletasked versions in the suite. This section also includes some observations on the iPSC/860 node performance.

2.1 Ground Rules

This work presents the changes required to convert single processor Y-MP vector code to iPSC/860 parallel, distributed memory code. The changes were made in the fashion of a tool-based conversion which preserved the original algorithm. While the performance of these codes do not represent the best iPSC/860 parallel implementation of the algorithms, a knowledge of the typical changes involved with such a conversion should allow NAS to judge the suitability of the various parallel tools for NAS users. Such knowledge should also help NAS influence the future development of such tools.

- Ground rules to guide the "tool-based" creation of double precision (64-bit) versions of all 5 programs in the suite include:
- Use of a manual port to permit the large-scale algorithm changes required by some codes to produce efficient performance.
- Use of natural (sequential) node ordering for mapping the two-dimensional grids onto the hypercube. This restriction means that nearest neighbor nodes in a grid layout will not correspond to nearest neighbors on the hypercube topology and that message-passing performance will suffer relative to a version employing Gray code numbering. This approach is consistent with the spirit of a tool-generated conversion.
- Emphasis on synchronous I/O for the message-passing. While it was thought initially that all codes could employ synchronous message-passing, later experience revealed that all codes could benefit from asynchronous data transmittal.
- Use only the iPSC/860 1MW per node memory. In some cases, this rule limits the suite problem sizes. In practice, iPSC/860 users must employ disk files or CFS to store data for large amounts of data. However, iPSC/860 I/O performance does not scale with the number of nodes (Nitzberg, 1993) and employing the CFS for extensive I/O would cloud the evaluation of the port.

- Use of limited compiler tuning. Except where noted, the only compiler tuning consisted of modifications to ensure that all major DO-LOOPS involved sequential access to the major arrays.

In addition, all programs used the iPSC/860 fast divide option and all program FLOPS reflect a manual FLOP count of the iPSC/860 source code with 4 FLOPS assigned for the divide. The divide count is somewhat lower than other estimates (Bailey, 1993), but the divide operation constitutes a small fraction of the overall FLOP count.

2.2 Evaluation Technique

The report provides several methods for evaluating the parallel decompositions. The first measure is the hardware efficiency, **e**, which is usually defined as

$$e = (T(1)) / (N \times T(N))$$

where **T(1)** and **T(N)** denote the elapsed time for execution of the same problem of 1 and N CPUs respectively. Parallel decomposition of the problem implies that each of the N nodes solves a smaller problem than that solved by a single node. Cache inefficiencies can inhibit such single node performance and lead to parallel efficiencies exceeding 100%. Thus, the report follows a previous approach (Dongarra, 1990) to define efficiency as

$$e = (R(1)) / (N \times R(N))$$

where **R(1)** denotes the maximum double precision single processor rate of 40 MFLOPS and **R(N)** denotes the N-processor rate.

A series of benchmark codes has rated the 128-node Intel Touchstone at about 250 MFLOPS on compiled code (Bailey, *et al.*, 1991). A second criterion for an effective port is that its 128-node performance should equal or exceed this 250 MFLOPS rating.

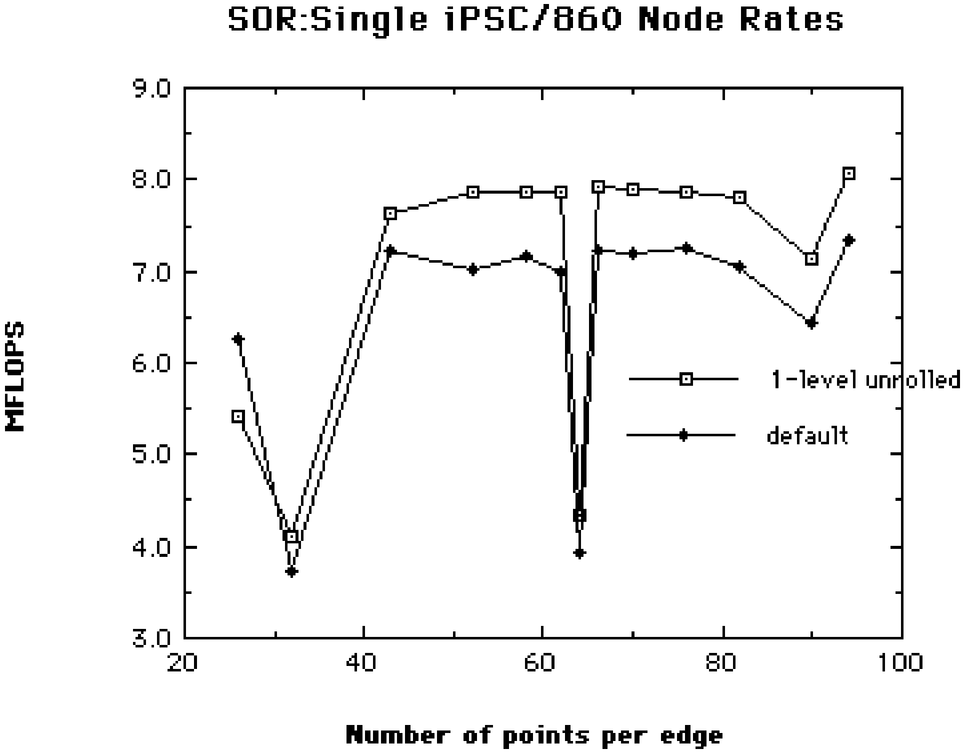
A third measure of the effectiveness of the iPSC/860 decompositions is the ratio of computation to communication and a ratio of 50% is reported as a relatively efficient implementation on a typical full-scale problem (Mavriplis, *et al.*, 1992).

2.3 Observations

The iPSC/860 chip demands careful attention to details for maximum performance. A 64-bit data bus connects the off-chip node memory to an on-chip data cache. Since the node memory is located off-chip, high-performance code must effectively utilize the small on-chip data cache and limited off-chip memory bandwidth (Lee, 1991). An onboard hardware cache supervisor manages the cache in real-time. Two programs producing essentially the same assembly code can display significantly different performance due to the intervention of this cache supervisor.

Supplying the compiler with code containing properly unrolled and/ or stripmined loops will improve cache utilization and application performance (Carr and Kennedy, 1992). These modifications enforce data locality, *i.e.*, they force the CPU to access data elements in terms of small neighborhoods of memory. Such access patterns promote the reuse of data stored in the on-chip data cache and prevent the CPU from stalling for lack of data. Performance is strongly sensitive to data location, *i.e.*, whether the application executes with data "in cache" or whether the application runs with data "out of cache". In some cases, the application performance degrades severely, as shown in Figure 1.

Figure 1



The figure shows severe performance degradation when the number of grid points per edge is a multiple of 32. The iPSC/860 node cache is 8K, two-way, set associative. The 8K denotes the size of the cache as 8K bytes or 1K (1024) double precision words. A two-way set associative cache places into the same cache set two words which are separated by a fixed multiple of words in the off-chip memory. In the i860 design, the fixed multiple is 4K bytes or 512 words. The algorithm for loading cache lines is random replacement. If the cache consists of two lines of contiguous data, it is possible for each data reference separated by 512 words to overwrite data before it is used. This separation can result in a constant loading/reloading (thrashing) of cache until the loop completes. Insertion of buffer arrays to separate critical arrays by more than 512 words should relieve the cache thrashing.

Figure 1 also shows typical single node performance and the sort of performance improvement which can be achieved due to loop unrolling. The computational loops were executed with stride 2 and the second iteration written out explicitly. This modification allowed a more efficient utilization of the on-chip cache and produced about a 10% improvement in performance.

The inability of the current iPSC/860 compiler to optimize compilations is due in part to compiler immaturity and also to the difficulty of generating efficient code for the iPSC/860 chip. The relatively small number of registers contained on the chip mandates a high degree of register reuse and forces an explicit control of the pipelined units (Case, 1992). Application of the Cray Fortran compiler expertise to the DEC ALPHA chip has currently yielded a performance improvement of about 50% relative to the native compiler and this value would seem to be a good estimate for the ability of a mature compiler on the iPSC/860.

3.0 Successive OverRelaxation Algorithm

3.1 Code Description

NAS users employ the Successive OverRelaxation (SOR) algorithm in advanced formulations involving fluid dynamics on unstructured grids and the aerodynamics of the Space Shuttle (NASA, 1990). In the parallel suite, the Y-MP single processor version of the SOR employed a Gauss-Seidel iteration on a cube geometry. As it travelled to each plane, the algorithm visited the grid points in a red/black checkerboard fashion to promote vectorization and applied Chebyshev acceleration factors to speed convergence. These factors involved the spectral radius of the Jacobi iteration, as estimated from the number of nodes in the grid (Press, *et al.*, 1986):

$$\mathrm{Rj=cos(pi/J)+(delx/dely)**2 * cos(pi/L)/(1+delx/dely)**2} \qquad (3)$$

where there were J by L nodes in the two-dimensional grid and for our problems, delx=dely. This formula strictly applies for a two-dimensional problem, and the 3D SOR employed this formulation of the acceleration factor with good results.

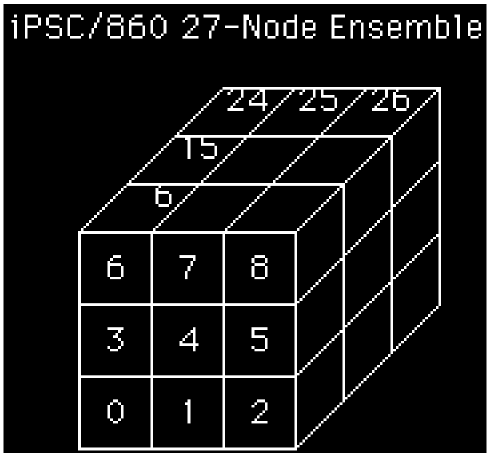
With some user intervention, the Cray autotasker, FPP (Cray, 1988) parallelized this calculation by allowing the CPUs to operate on the red points on all planes and then on the black points on all planes. FPP extracted essentially all of the parallelism available at a cube size of 256 and FPP’s version of the code performed at over 80% efficiency on an 8-CPU Y-MP. Measurements of code performance indicated that the large problems were able to amortize the overhead from load imbalance and delay from memory conflicts.

3.2 Modifications for Parallel Execution on the Hypercube

The initial iPSC/860 version of the SOR maintained the FPP planar parallelism, *i.e.*, each node of the iPSC/860 updated a single plane while maintaining the red/black ordering. Each node employed synchronous message passing to transmit its values as a boundary condition to the adjacent nodes. The nodes then checked for global convergence before initiating a new iteration. Performance on a 128-node problem, with each node calculating on a 128 by 128 point square was only 35 MFLOPS. Allowing each node to treat several planes improved the performance slightly to 36 MFLOPS.

A decomposition with an improved ratio of computation to communication (Fox, *et al.*, 1988) motivated a second implementation of the SOR. The large memory bandwidth of the Y-MP permitted efficient execution of the "cube of planes" decomposition, but the smaller memory bandwidth of the hypercube required a "cube of cubes" ensemble as shown in Figure 2.

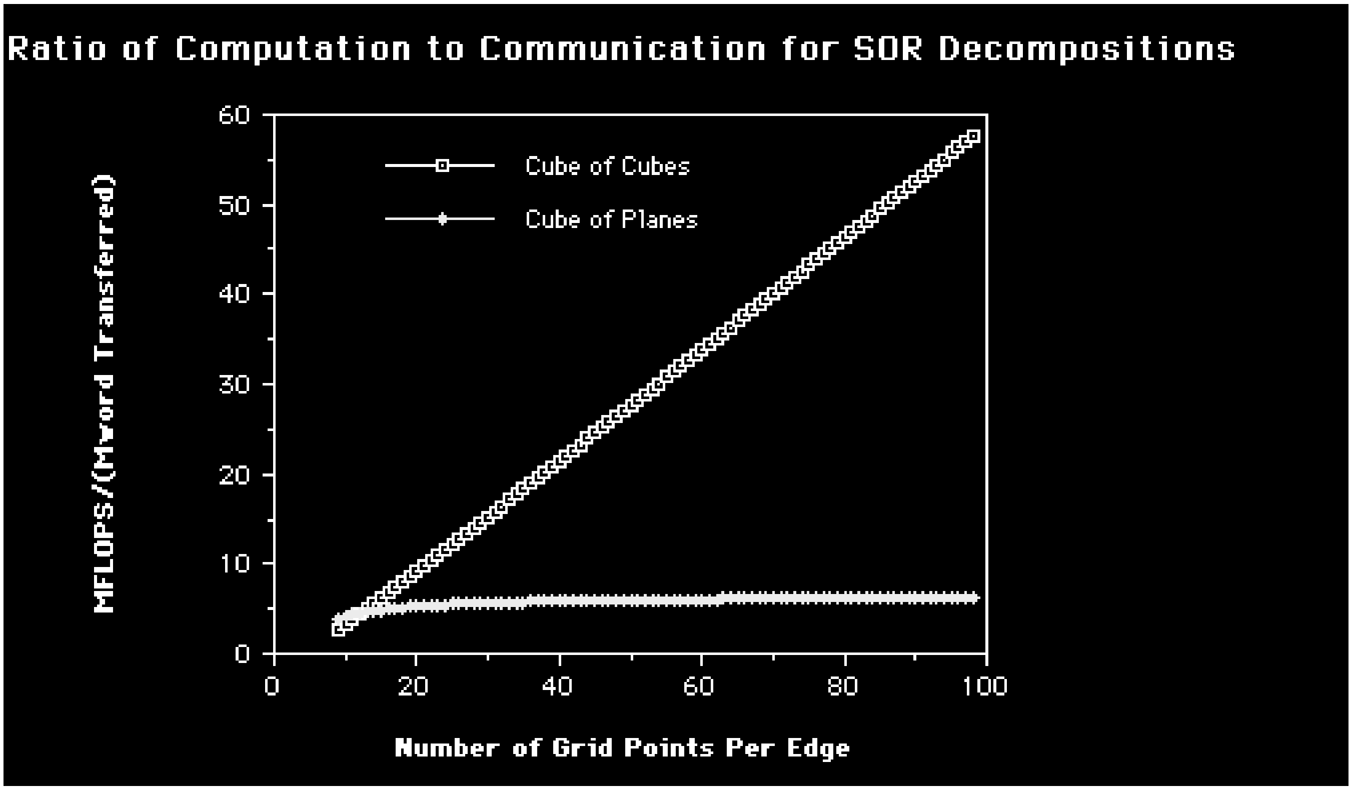
Figure 2



The amount of computation depends upon the volume enclosed by the component cube while the amount of data transmitted to neighboring component cubes depends upon the surface area of the individual component cubes. The improved decomposition has a larger volume to surface-area ratio than the planar decomposition, *i.e.*, it performs a larger amount of computation for a given amount of data transfer.

A comparison of the ratio of the computation load to the communication load allows rough estimate of the effectiveness of the two decompositions. The computation load is the total number of floating point operations performed by all nodes in an iteration and the communication load is simply the total number of 8-byte words transferred by all nodes in an iteration. These ratios are easy to establish for the two algorithms since the code is compact. Figure 3 displays the ratio (MFLOPS per Mword) for two decompositions, one consisting of a cube of cubes and the other consisting of a cube of planes. The X-axis denotes the number of grid points on an edge for each of the nodes in the decomposition. Relative to the cube of planes decomposition, Figure 3 shows that the cube of cubes decomposition performs a greater amount of calculation for each word of data transferred .

Figure 3



The improved decomposition, in which each cube generally has 6 neighboring cubes, required each cube to identify the node numbers of its nearest neighbors to ensure proper message-passing. The messages consisted of the values in the calculated planes nearest the adjacent node. These values were exchanged as boundary values with the neighboring node. The decomposition is termed a simple domain decomposition with overlapping boundary values.

The above decomposition admits a variety of parallel SOR implementations. One hypercube implementation of the red/black algorithm labels half of the iPSC nodes red and the other half black while a second implementation labels grid points in the individual iPSC nodes as red and black. Since the ordering requirement for the first implementation dictates half-iteration on the red nodes and half-iteration on the black nodes, half of the nodes would be computationally idle during the iteration. The second implementation requires that each iPSC node visit half of its grid points during the red iteration and half of its grid points during the black iteration, and all iPSC nodes would be working during the iteration. Both implementations would require one synchronization and boundary exchange after the red node update and one synchronization and boundary exchange after the black node update.

Idle processor considerations may also make attractive a third implementation, the substitution of a simultaneous update scheme, *i.e.*, a Jacobi iteration. This change would reduce the message-passing and keep the hypercube nodes busy during most of the problem. This change would also slow the convergence of the iterations because acceleration factors would no longer apply.

A desire to maintain convergence while employing a tool-oriented transformation dictated the selection of the first method for this report. The first method, employing the global red/black algorithm, seemed most likely to be achieved by a parallel tool since it is the closest in spirit to the Y-MP version. The nodal red/black version requires detailed bookkeeping, although it may be a more efficient technique. Substitution of the Jacobi iteration for the red/black technique would reduce the rate of convergence. In fact, parallel decomposition can strongly effect the rates of convergence for the SOR algorithm (Adams and Jordan, 1986). Since this report emphasizes issues involved in an automated port of vectorized code to a parallel machine, convergence questions will be deferred as beyond the scope of the present work.

An attempt to maintain consistency among the variously-sized ensemble calculations required a modification to the termination criterion. The Y-MP version used the reduction of a global norm below an input value to signal the end of the calculation. For the hypercube, this criterion does not ensure that average values on each node are close enough to the desired values. Instead, the hypercube versions require that the average value in the cube approach an input value.

3.3 Code Performance

This section discusses the parallel performance of SOR algorithm on the iPSC/860. Table 1 shows performance as a function of problem size for acceleration factors determined by the prescription given above. The table includes extra cube sizes for the 64-node ensembles since the regular sizes of 32 and 64 suffers severe performance degradation from cache-thrashing.

Table 1

Red/Black SOR iPSC/860 Performance 1283 Problem**

Ensemble	Cube Size	Iterations	Telapsed	Tcomm	MFLOPS	Efficiency
27	43	192	64.3	40.6	83.4	0.078
64	32	175	43.1	26.3	105.2	0.041
64	34	184	33.3	21.9	173.7	0.068
125	26	168	16.3	11.6	267.1	0.053

2563 Problem**

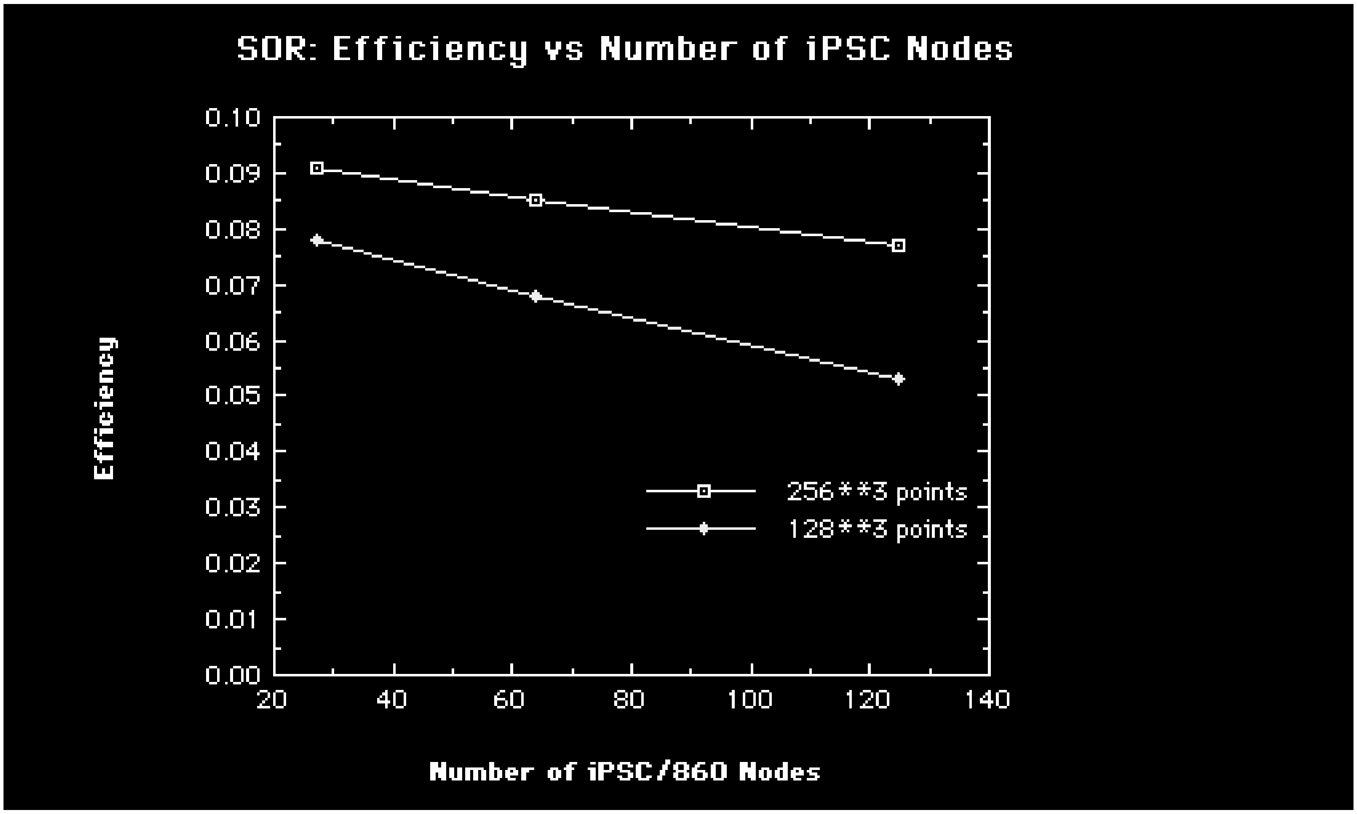
Ensemble	Cube Size	Iterations	Telapsed	Tcomm	MFLOPS	Efficiency
27	86	353	866.7	493.3	97.8	0.091
64	64	316	594.6	335.4	121.6	0.048
64	66	324	375.6	221.9	217.1	0.085
125	52	292	178.9	114.5	382.4	0.077

The decreases in elapsed time indicate that the decomposition has allowed the effective application of an increasing number of nodes. The column labelled "Tcomm" represents the time spent in the communication routines and the global sum. The ensembles spend more than 50% of the elapsed time in the communication routines and the dominant amount of this time is spent blocking while waiting to receive a message. Dividing the total problem floating point operations by the time spent in computation only gives computation rates averaging between 5.3 to 8.5 MFLOPS per node. The measured ratio of floating point operations to data words sent via message-passing ranged from 19.3 for the 27-node 256**3 problem to 3.0 for the 125-node 128**3 problem. The Performance Analysis Tool (PAT) indicated that this decomposition allowed a balanced computational load across all processors with about 40% of the elapsed time spent in computation.

The table shows that, for a given problem size, an ensemble composed of a larger number of cubes requires less iterations than an ensemble composed of a smaller number of cubes. This effect is counter-intuitive since an increased number of cubes in an ensemble implies a greater decoupling of the problem. The intuitive viewpoint neglects the powerful influence of the Chebyshev acceleration factors. The problem-dependent acceleration factors provided by the spectral radius (3) are not optimal. Testing of the formula on single-cube problems indicated that the spectral radius approach overestimates the optimal factors and that the overestimate is larger for larger cube sizes. The effect of the acceleration factors exceeds the decoupling brought about by decomposition. The iteration data illustrate the importance of understanding algorithm convergence properties before attempting a parallel decomposition.

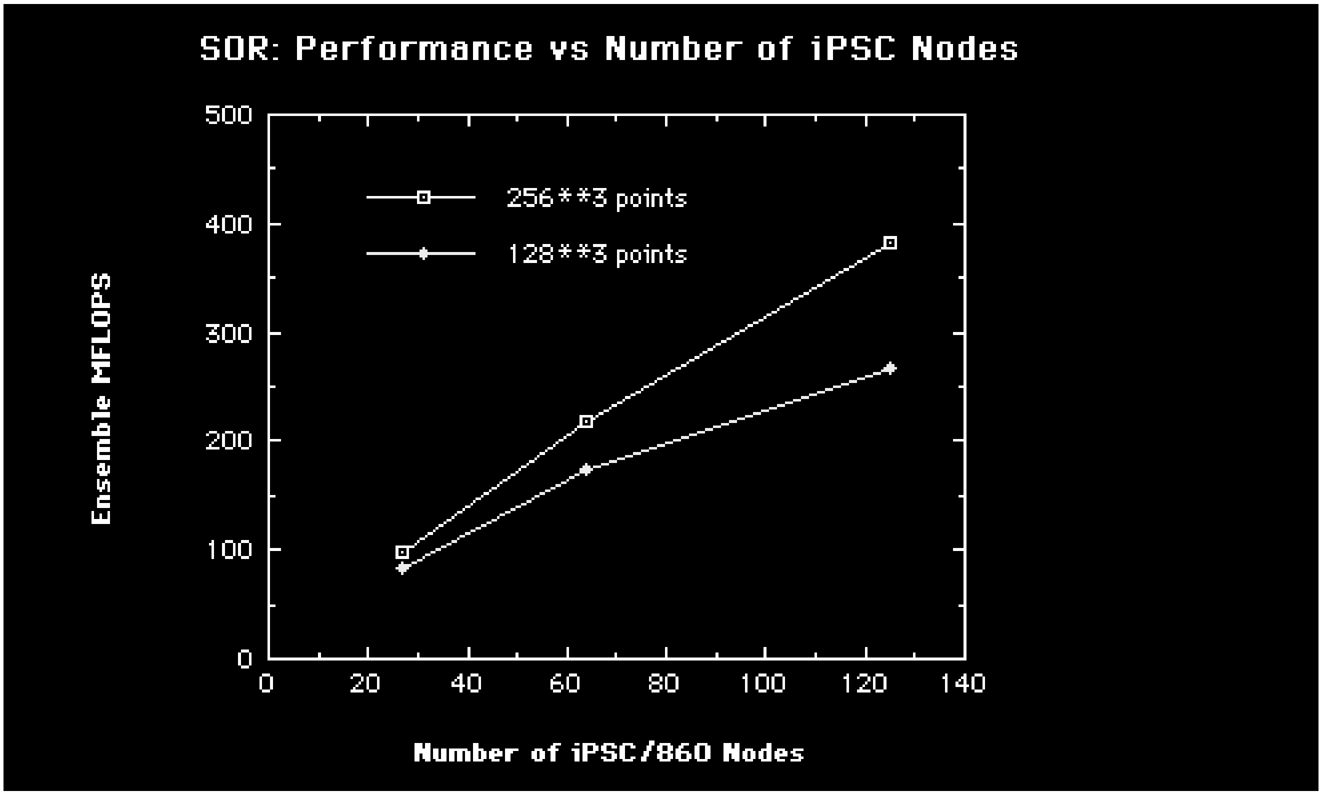
Machine efficiencies, computed according to prescription given in Section 2.3, lie in the 5-10% range as shown in Figure 4.

Figure 4



One way to assess the decomposition is to examine how it scales. Figure 5 shows the ensemble MFLOP rates as a function of increasing number of nodes.

Figure 5



Since the number of grid points assigned to each iPSC/860 node have been chosen to conserve the total number of mesh points in the ensemble, the amount of floating point computation per node per iteration remains roughly constant. If there is no increasing loss to communication time as the problem size increases, the increase in ensemble MFLOP rate should be linear with the number of processors. Figure 5 shows a somewhat less-than-linear increase in performance. Although this algorithm is nearest-neighbor, most of the problem time is spent in communication and the bulk of this time is in the message receive (crecv) routines, where the node must wait, or block, until it receives a transmitted message. Only half of the nodes are calculating during the iteration and this behavior seems to contribute the the poor communication scaling. An obvious improvement to this decomposition would modify the algorithm to permit asynchronous communication.

Overall, the measurements indicates that the decomposition produced a reasonable hypercube performance. Employing an 86**3 grid on the 125 node ensemble yields a performance of 430 MFLOPS.

3.4 Lessons for Tools

The most important factor in porting the SOR to the iPSC/860 is a decomposition which maximizes the ratio of volume to surface area. This factor improved performance by a factor of 8 over the Y-MP memory-intensive decomposition. This approach automatically results in good data locality and favorable scaling characteristics. The decomposition employs no Fortran extensions beyond those required for nearest-neighbor message-passing and global sums.

To employ a cube of cubes decomposition, an automatic tool would have to obtain the dimensionality of the problem from the user since the Fortran implementation here uses two-dimensional arrays to describe a three-dimensional geometry. The tool would have to color the nodes quite carefully (again, with assistance from the user) to ensure the opposite colors are the nearest neighbors in the three-dimensional scheme. Application of red/black noding to the individual nodes comprising the hypercube would allow all nodes to compute during the iteration, but would require a more complicated labelling arrangement to carry out the red/black ordering.

The key computations, carried out in the two DO-loops performing the red and black global sweeps, require transmittal of data to the neighboring nodes after the execution of the DO-loops.

4.0 Shallow Water Model

4.1 Code Description

Many NAS users employ explicit timestepping of CFD equations treating rate-dependent phenomena, such as combustion and other chemical reactions (NASA, 1990). The finite difference equations for the shallow water model (SWM) also employ explicit timestepping and contain a space discretization based on Taylor expansions. The two-dimensional SWM model, representing computations employed in atmospheric modelling, is a system of three equations in three unknowns, the x-velocity, the y-velocity and the height of the fluid. The current formulation as presented by Sadourny (1975) and implemented by Hoffman, *et al.* (1986), conserves the mean square vorticity, or enstrophy. Application of periodic boundary conditions and use of a regular geometry introduced a high degree of parallelism into the code.

The previous report discussed the FPP implementation of fine-grained parallelism through its creation of 3 parallel regions corresponding to the major computational loops. CPU synchronization occurred after completion of each DO-loop. The parallel version displayed excellent Y-MP performance as this approach allowed almost perfect CPU load balancing with each CPU executing a long-vector length computational workload. The FPP-generated version displayed an efficiency exceeding 0.9. Performance measurements indicated a slight load imbalance generated by singletasked regions which implemented the periodic boundary conditions. As the number of CPUs increased and the elapsed time decreased, this load imbalance exerted a greater influence because the fraction of time spent in singletasked mode increased.

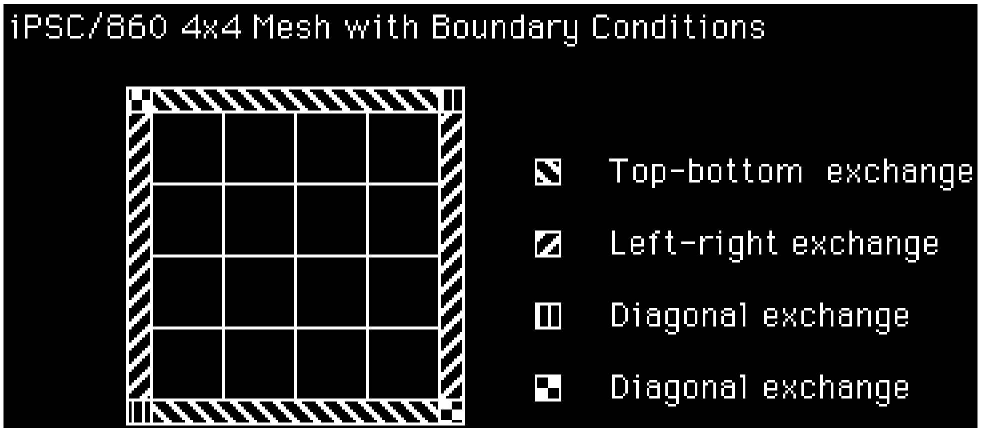
4.2 Modifications for Parallel Execution on the Hypercube

Geometry provided the basis for the high-level decomposition in that each processor would work on a square subset of the global square. Given this decomposition, the most obvious way to parallelize SWM was to parallelize the three major DO-loops just as FPP had done on the Y-MP.

Decomposition into a square of squares required a two-dimensional version of the nearest-neighbor message-passing routine described in section 3.2. After each of the three double DO-loops, transmittal of newly calculated pressures and velocities to remote-neighbors satisfied the doubly periodic boundary condition. An additional data transfer to nearest-neighbors satisfied the continuity requirements for the decomposition.

Figure 6 shows the boundary transfers for a schematic 4 by 4 mesh. The shaded regions represent outer rows or points of the boundary nodes.

Figure 6



All interior nodes send and receive messages from their 4 nearest neighbors. Nodes containing the boundary points send and receive messages from their nearest neighbors and also send and receive messages from the appropriate remote nodes. Figure 6 shows the types of remote neighbor exchanges required by the boundary conditions. Processor nodes in the top row of the mesh compute values on grids, which contain the boundary points as their top rows. These nodes must exchange their boundary values with the boundary values of nodes on the bottom of the 4 by 4 mesh. Similar exchanges occur for nodes on the left and right boundary. Diagonal exchanges involve values computed for the extreme points (only) on the opposite diagonals of the mesh.

An initial 2-node iPSC/860 version of the SWM ensured that the task decomposition performed correctly. Since the general problem would involve many squares, some additional code was inserted to ensure that global momentum and enstrophy were conserved. A second 4-node version ensured that data transfers in all 4 directions executed correctly. Decomposition of the general problem into a square of squares on the iPSC/860 prevented the exact duplication of the Y-MP results. The discrepancy arose because each iPSC/860 square required its own boundary nodes and the single-square Y-MP had boundary nodes at only 4 edges. Global momentum and enstrophy differed by less than one percent between the Y-MP and iPSC/860 versions.

4.3 Code Performance

This section discusses the parallel performance of SWM performance on the iPSC/860. Table 2 shows performance as a function of problem size for 200 timesteps.

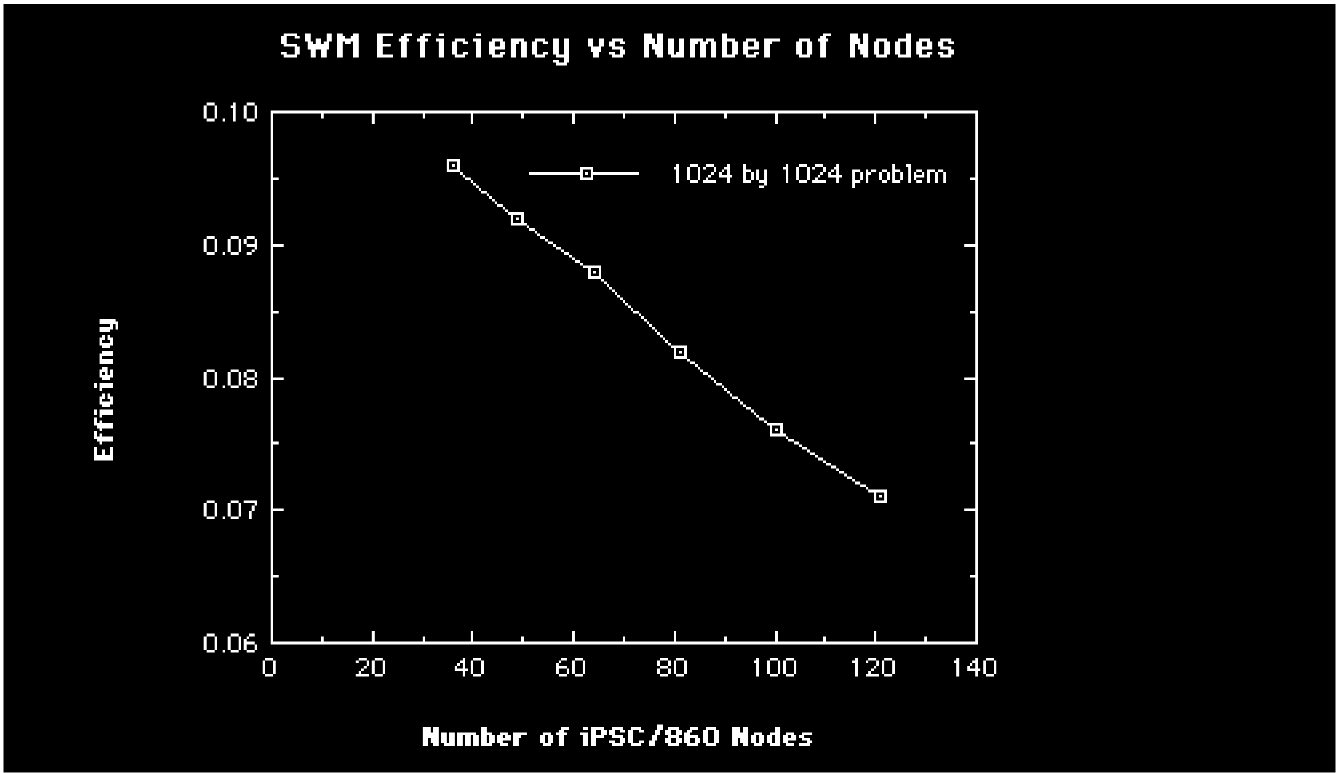
Table 2
SWM iPSC/860 Performance on 1024**2 Problem

Ensem	Grid	Telaps	Tcomp	Tcom	MFLO	Effm
ble		ed		m	PS	
36	171	98.2	84.2	14.0	139.4	0.096
49	147	76.5	62.5	14.0	179.9	0.092
64	128	60.4	47.4	13.0	225.8	0.088
81	114	51.5	37.5	14.0	265.7	0.082
100	102	44.3	30.3	14.0	305.3	0.076
121	94	40.5	25.9	14.6	343.5	0.071

The number of grid points per edge decreases as the number of squares in the ensemble increases to maintain the same overall problem size. As the number of nodes increases, the time spent on computation decreases and the time spent on communication remains approximately constant. This observation would appear to apply to all nearest-neighbor type algorithms and accounts for their effectiveness on highly parallel machines. The time spent in the periodic boundary condition message-passing contributes about 60% of the total communication time. The measured ratio of floating point operations to data words sent via message-passing ranged from 15 for the 16-node problem to 8 for the 121-node problem. The Performance Analysis Tool (PAT) indicated that the square of squares decomposition allowed a balanced computational load across all processors with about 80% of the elapsed time spent in computation.

Figure 7 shows the machine efficiencies for the SWM application.

Figure 7



The machine efficiencies are in the same range as the SOR problem discussed in Section 3 and decrease to 7% for the 121 node case.

Figure 8 shows the scaling behavior of the SWM ensemble MFLOP rates as a function of the number of iPSC nodes.

Figure 8

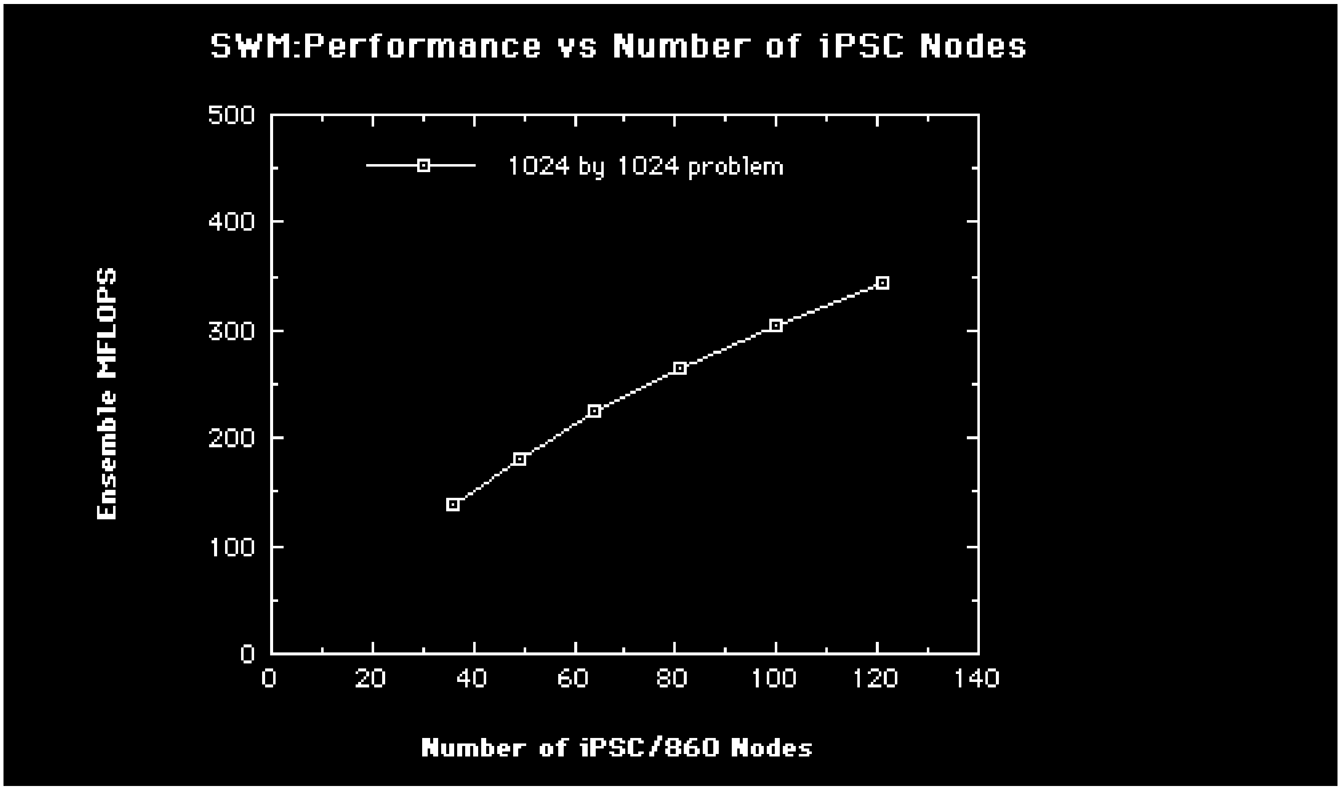


Figure 8 indicates a slight departure from a linear increase in performance. This behavior is expected since, with the increasing number of nodes, the time spent in computation decreases and the time spent in communication remains constant.

4.4 Lessons for Tools

The most important factor in porting the SWM to the iPSC/860 is a decomposition which maximizes the ratio of the surface area to the perimeter for the individual hypercube nodes. The key computations, carried out in the three DO-loops performing the global sweeps, require transmittal of data to treat the periodic boundary conditions after their execution. Most of the porting effort consisted of constructing the routines to enforce the periodic boundary conditions. While the square of squares decomposition allowed easy expression of these conditions, these relations involved 43 separate equations.

5.0 Tridiagonal Solver—

5.1 Code Description

Tridiagonal systems occur repeatedly in finite-difference approximations to differential equations with 2nd-order derivatives. The Y-MP version of the tridiagonal solver employed the parallel cyclic reduction technique (Hockney and Jesshope, 1988), a highly effective method for solving tridiagonal systems on a vector computer and on a multiprocessor.

The cyclic reduction method consists of a series of forward iterations, which eliminate the odd-numbered equations. The procedure finally arrives at a single equation; the algorithm then backsubstitutes through the reduced systems until the original set of equations is solved.

The Y-MP version performed all phases of the reduction and backsubstitution in parallel. The forward iteration involves the recursive calculation of data on both the left and right sides of the equations. Each step of the reduction accesses memory locations in a nonlinear manner (skips through memory) because the number of equations decreases by a factor of 2 in each reduction. The Y-MP implementation exploited the shared memory to perform the algorithm's non-sequential memory access by careful indexing. Every CPU was able to access the data and the large system size amortized the delay due to shared memory conflicts.

5.2 Modifications for Parallel Execution on the Hypercube

The initial implementation of the cyclic reduction algorithm attempted to mimic the shared memory version by using a broadcast approach (sending all data to all processors). The ratio of computation to communication for this algorithm was extremely low, indicating the algorithm was communication-bound. Reimplementation of cyclic reduction using a pipelined technique with special coding to maximize nearest-neighbor communication provided one option. However, the partitioned vectorized solver (Wang, 1981) appeared more attractive due to ease of implementation and low interprocessor communication requirements. This solver partitioned the tridiagonal matrix into a set of square blocks and then applied elementary row transformations to obtain a structure with only one "independent" variable nested in each partition. Figure 9 shows the matrix structure of a 16x16 system of equations after the application of the row transformations. This structure now consists of 4 systems (denoted by the horizontal layers) for execution on 4 processors.

Figure 9
Transformed Partition of 16x16 Tridiagonal Matrix

$$
\begin{array}{cccc|cccc|cccc|cccc|c}
a01 & & & g02 & & & & & & & & & & & & & x01 \\
& a02 & & g02 & & & & & & & & & & & & & x02 \\
& & a03 & g03 & & & & & & & & & & & & & x03 \\
& & & a04 & & & & g04 & & & & & & & & & x04\ * \\
\hline
& & & c05 & a05 & & & g05 & & & & & & & & & x05 \\
& & & f06 & & a06 & & g06 & & & & & & & & & x06 \\
& & & f07 & & & a07 & g07 & & & & & & & & & x07 \\
& & & f08 & & & & a08 & & & & g08 & & & & & x08\ * \\
\hline
& & & & & & & c09 & a09 & & & g09 & & & & & x09 \\
& & & & & & & f10 & & a10 & & g10 & & & & & x10 \\
& & & & & & & f11 & & & a11 & b11 & & & & & x11 \\
& & & & & & & f12 & & & & a12 & & & & g12 & x12\ * \\
\hline
& & & & & & & & & & & c13 & a13 & & & g13 & x13 \\
& & & & & & & & & & & f14 & & a14 & & g14 & x14 \\
& & & & & & & & & & & f15 & & & a15 & b15 & x15 \\
& & & & & & & & & & & f16 & & & & a16 & x16\ * \\
\hline
\end{array}
$$

The figure also shows that the row transformations have reduced the system to 4 independent equations, denoted by asterisks. A Gaussian algorithm performed by a single processor (node 0) can solve the reduced tridiagonal system for the unknowns x04, x08, x12, and x16. Each processor sends only 4 words to the master processor solving the reduced system and receives only 4 words from the master. Backsubstitution, performed in parallel, provides the solution to the remaining 12 unknowns.

The 4-word transmittal requirement is independent of the number of equations in the block and this feature makes the algorithm scalable for systems with many processors. On Cray architectures, however, the partitioned solver does not parallelize well. The algorithm requires that the outer loop denote the number of equations in the system and the inner loop denote the number of partitions. This arrangement limits the amount of work given to each Cray processor.

5.3 Code Performance

Table 3 shows ensemble performance as a function of matrix size. The base case treats a 2**16 linear system, *i.e.*, a system of 65536 coupled equations. For comparison, a typical two-dimensional CFD problem (100 by 100 with 4 unknowns per node) has about 40,000 unknowns. The table also shows the performance of two larger systems to show the affect of additional work on the performance of the algorithm. For 100 passes through the solver, single processor performance data are as follows:

Table 3
Partitioned Tridiagonal Solver- iPSC/860 Performance
2**16 Linear System

Ensemble	Telapsed	Tcomp	Tcomm	MFLOPS	Effm
8	5.73	5.72	0.10	32.1	0.100
16	3.00	2.71	0.28	61.4	0.096
32	1.76	1.29	0.47	104.5	0.082
64	1.37	0.34	1.03	133.9	0.052
128	2.42	0.41	2.01	75.6	0.015

2**17 Linear System

Ensemble	Telapsed	Tcomp	Tcomm	MFLOPS	Effm
8	11.38	11.28	0.10	32.4	0.101
16	5.85	5.60	0.25	62.7	0.098
32	3.18	2.71	0.47	115.3	0.090
64	2.12	1.20	0.92	172.3	0.067
128	2.76	0.95	1.81	132.5	0.026

2**18 Linear System

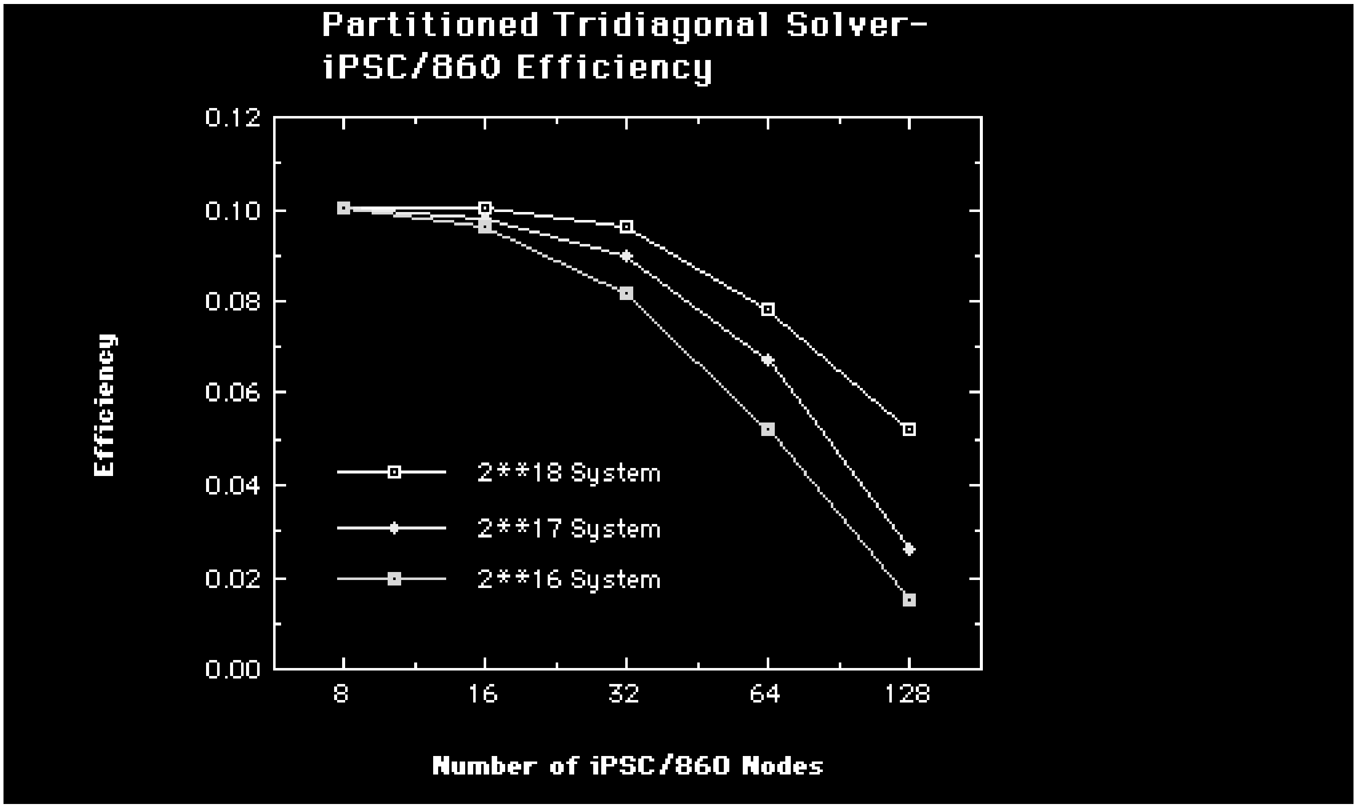
Ensemble	Telapsed	Tcomp	Tcomm	MFLOPS	Effm
8	22.69	22.59	0.10	32.4	0.100
16	11.51	11.02	0.49	63.8	0.100
32	5.98	5.27	0.71	122.7	0.096
64	3.67	2.76	0.91	199.7	0.078
128	2.78	2.07	0.71	263.8	0.052

The table shows that increased processors tend to reduce and then increase elapsed times for each system. Generally, additional processors decrease computation times and increase communication times. Analysis of the time required for arithmetic operations and communication yields an algebraic expression for the total time required for the parallel computation (Johnson, *et al.*, 1987). For a fixed problem size, evaluation of the formula as a function of the number of processors indicates a minimum total time. The cost of the forward and backward eliminations decreases with additional processors, but the Gaussian system which must be solved by a single processor grows with increasing processors. Communication costs also increase with increasing processors.

The measured ratio of floating point operations to data words sent via message-passing ranged from 1250 for the 8-node 256**3 problem to 16 for the 128-node 256**3 problem. For the larger systems, PAT indicated that this algorithm overloaded node 0 with the Gaussian elimination computation. Application of the partition algorithm recursively to the Gaussian elimination would reduce this imbalance. The average processor spent about 10% of its elapsed time in computation.

Figure 10 shows the efficiency as a function of the number of processors.

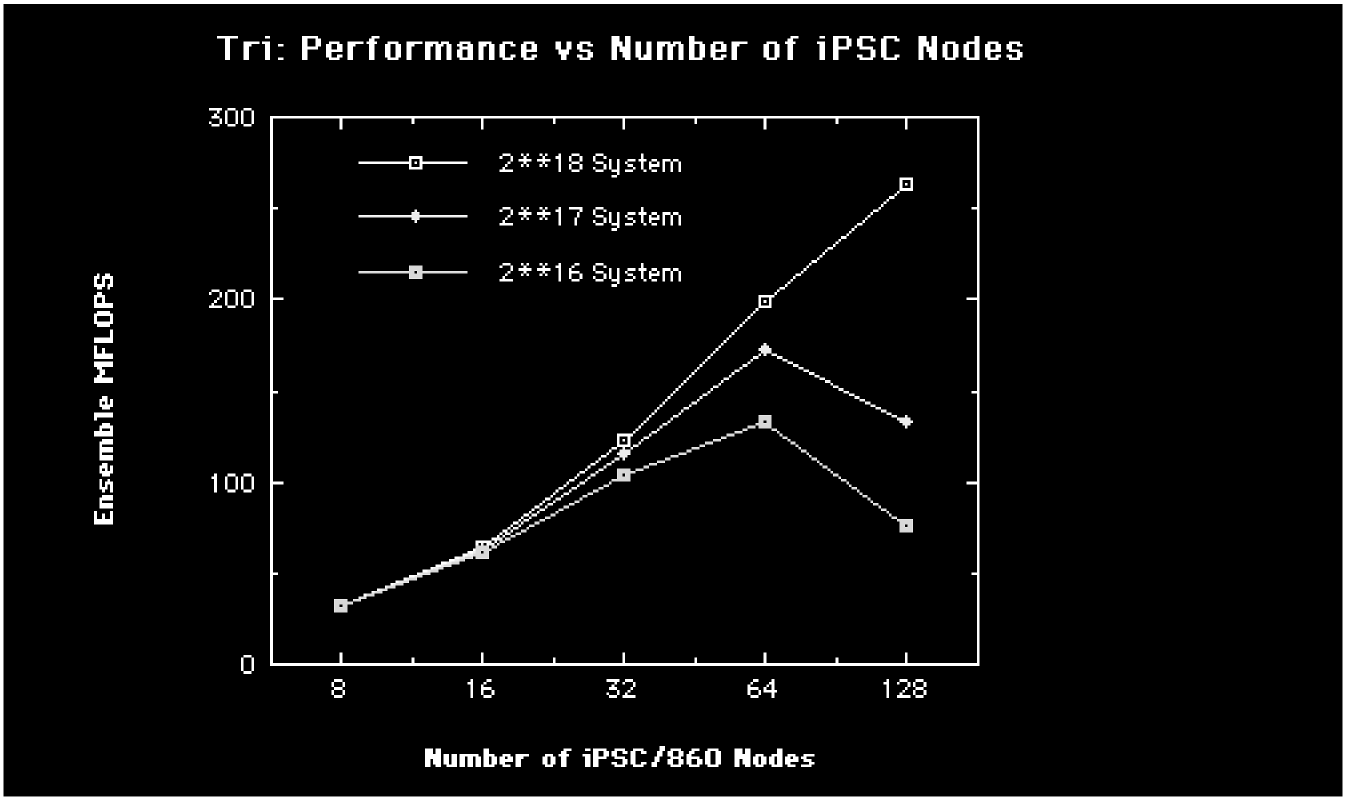
Figure 10



For two-dimensional systems, implementation of the NASPACK (Lee, 1991) tridiagonal solver would increase efficiency by decreasing the time spent by the single node in solving the reduced system.

Figure 11 shows the performance as a function of number of processors.

Figure 11



The increased communication costs combine with the cost of solving a large system on single node to produce a peak in ensemble performance. The figure suggests that this peak has yet to occur for the 2**18 system.

5.4 Lessons for Tools

The most important factor in porting the tridiagonal solver to the iPSC/860 was the ratio of computation to communication. The decomposition employed for the partitioned solver was algorithmic, whereas the previous two sections employed decompositions based on geometry. In all cases, an approach which maximizes the above ratio was the key feature contributing to the effectiveness of the hypercube decompositions.

Tools should assist the user in evaluating the computation to communication ratio. If this ratio is low, the tool can at least help to inform the user that the algorithm needs revision. While it may be unlikely for a tool to provide custom solvers, a site may be able to assist its users by providing examples of such solvers.

6.0 Alternating Direction Implicit Algorithm

6.1 Code Description

The Alternating Direction Implicit (ADI) method forms an important class of solution procedures on the NAS machines, especially in the field of global circulation models (NASA, 1990). The ADI algorithm employs an operator splitting technique to decompose the problem into multiple one-dimensional subproblems. The algorithm used in the Y-MP port followed a standard, modular implementation (Press, *et al.*, 1986), replacing the Gaussian solver described therein by a

power-of-2 cyclic reduction method to solve the tridiagonal systems.

The Cray autotasker, FPP, constructed a parallel version from the vector version described above by executing the assembly and backsubstitution loops of the x-sweep and the y-sweep in parallel. Calculations in the cyclic tridiagonal solver were also performed in parallel. Even with the tridiagonal solver brought into the main routine, FPP could not recognize that the entire operation, *i.e.*, assembly, solution, and backsubstitution, for each column of the x-sweep and each row of the y-sweep could execute in parallel. The tool uncovered useful parallelism only in the solver.

6.2 Modifications for Parallel Execution on the Hypercube

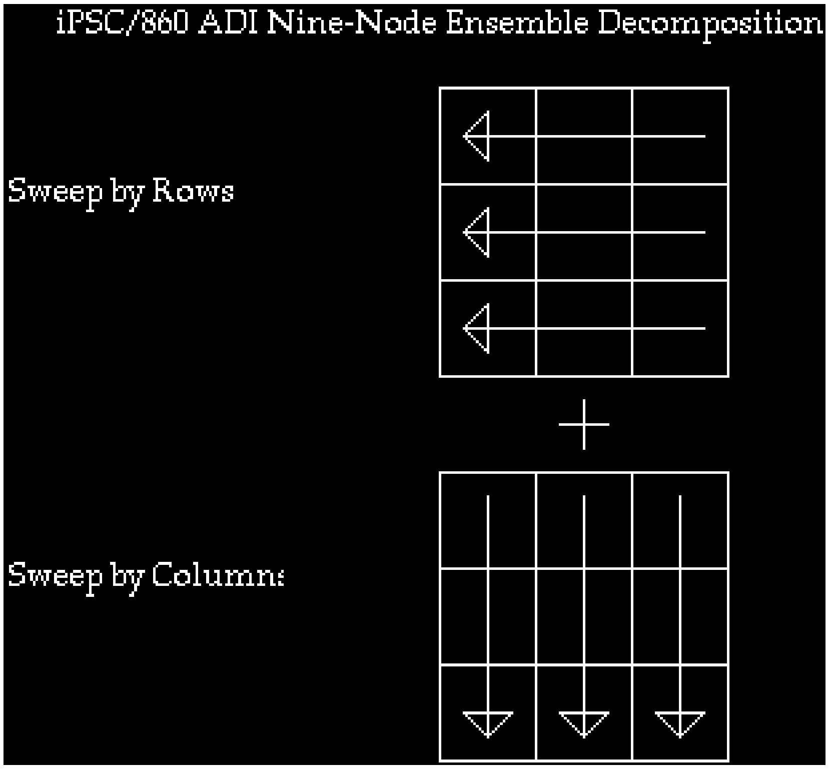
Experience with the hypercube cyclic reduction algorithm, as described in Section 5, led to the adoption of the partitioned solver as a more efficient tridiagonal solver. Insertion of this solver into the ADI in the standard manner of passing single rows or planes to the solver in a synchronous fashion gave poor performance.

An analysis of ADI methods recommended pipelining the algorithm to improve performance (Johnson, *et al.*, 1987). A pipelined algorithm consists of a sequence of stages and has the property that new operations can be initiated at the start of the pipeline while other operations are in progress through the pipeline. Compilers commonly apply pipelining to optimization of loops on vector architectures (Stone, 1987).

Pipelining the ADI method requires that all parts of the algorithm, assembling the left and right hand sides, the solver, and the backsubstitution execute in a stepwise fashion. The method requires a partitioning of the main computational loops into a set of tasks which can begin one after another before the previous task has completed. Creating such a partition requires a fairly strong understanding of the various dependencies in the algorithm.

Two partitions were employed in the ADI algorithm. The partitioned tridiagonal solver dictated a domain decomposition involving the ensemble grid. The desire to amortize communication overhead dictated the second partition involving a coarse-grained division of the nodal mesh into blocks or slices. Figure 12 illustrates first partition and the nodal data transmittal for the ADI iteration which does an x-sweep and a y-sweep on a 3 by 3 mesh.

Figure 12



In the first stage of the x-sweep (sweep by rows), each of the nodes in the same row sends the results of its forward elimination to the node immediately to its left. At a later stage, each of the nodes in the same row sends the results of its backward eliminations to the x-sweep "driver" nodes, which are located in the left column.

In the first stage of the y-sweep (sweep by columns), each of the nodes in the same column sends the results of its forward elimination to the node immediately below it. At a later stage, each of the nodes in the same column sends the results of its backward eliminations to the y-sweep "driver" nodes, which are located in the bottom row.

An ADI iteration can employ the above decomposition and synchronous data communication while sweeping across single rows or columns of each nodal mesh. Performance results for this strategy, defined here as a synchronous fine-grained iteration, are shown in the next section.

However, efficient hypercube pipelining requires asynchronous communication. In the pipelined ADI scheme, the node can post a receive request and then do some more work, such as assembling the next right and left-hand side, before using the data required by the receive request. In this way, computation overlaps with communication. Implementation of this scheme must contain a sufficient amount of computation to amortize the overhead of communication, *i.e.*, the implementation should be coarse-grained.

Figure 13 illustrates the second partition, the division of the x-sweep nodal calculation into blocks. The figure shows a mesh requiring 4 passes to complete a single x-sweep iteration. Each pass includes block 1 and block 2, and each of these blocks may contain a single row or a group of rows. Double-buffering refers to the requirement of separate temporary storage areas for block 1 and block 2. Testing determined that a two-pass partition provided the maximum performance for these problems; thus, each of the blocks operated on one-fourth of the mesh.

Figure 13

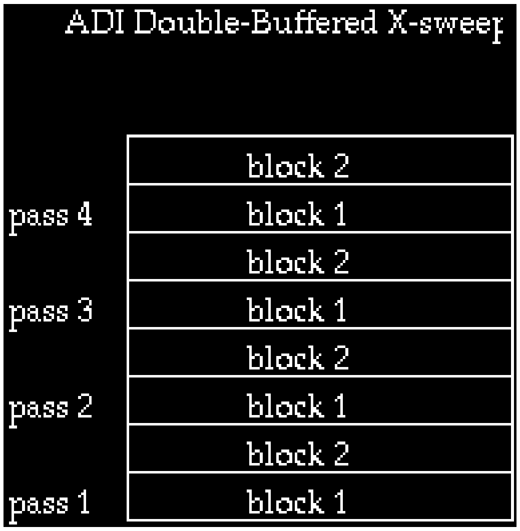
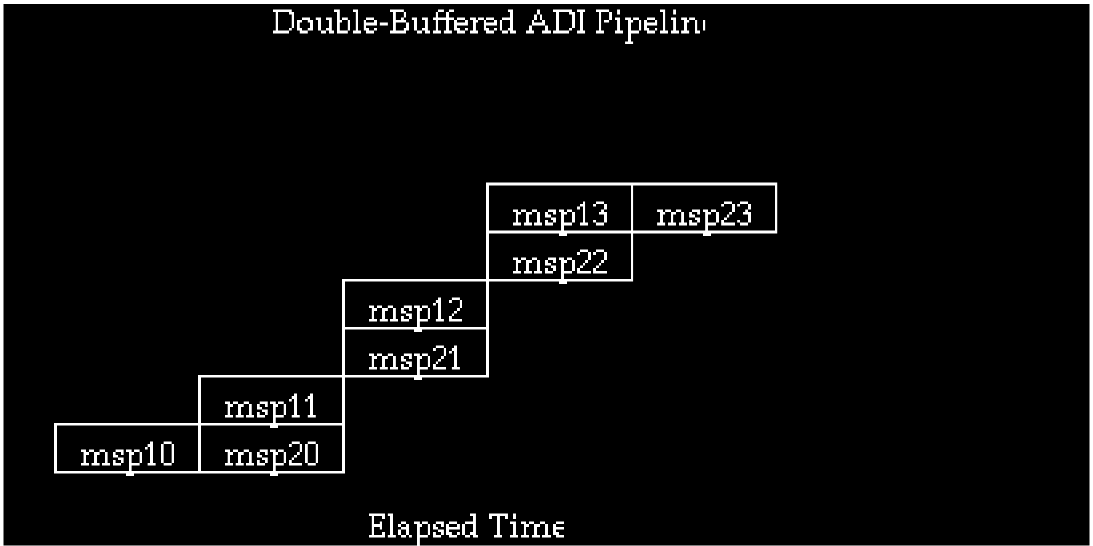


Figure 14 illustrates the approach to pipelining the nodal calculations. The ADI algorithm consisted of 4 separate stages (denoted as msp for multistage pipeline) as shown in the figure.

Figure 14



msp10 & msp20—assembly, forward elimination, and transmit to neighbor
msp11 & msp21—receive, backward elimination, and transmit to driver
msp12 & msp22—driver nodes—solve and transmit; other nodes—post receive
msp 13 & msp23—driver nodes—solve; other nodes—receive data and backsolve

Since the algorithm consists of 4 separate stages, it would be possible to employ a 4-way buffer on the problem, but the limited time spent in communication as shown in the next section seems to indicate that the additional stages would incur little performance benefit.

6.3 Code Performance

The base case treats a square with 1024 nodes per edge to give about 1 million points per grid. Table 4 shows performance data for the synchronous and asynchronous ADI versions. As with the SOR discussed in Section 3, acceleration factors play an important role in the rate of convergence and use of the same factor for all the ensemble sizes gave a consistent number of iterations.

Table 4
ADI iPSC/860 Performance on 128**3 Problem
Synchronous---Fine-grained

Ensemble	Grid	Telapsed	Tcomp	Tcomm	MFLOPS	Effm
16	256	283.3	255.6	27.7	41.1	0.064
25	206	199.6	177.9	21.7	58.8	0.059
36	172	158.4	140.8	17.6	74.0	0.051
49	148	129.5	115.0	14.5	90.8	0.046
64	128	106.7	95.2	11.5	104.9	0.041
81	114	102.4	90.4	12.0	109.7	0.034
100	100	95.4	85.4	10.0	122.7	0.031
121	94	88.1	79.3	8.8	130.7	0.027

Asynchronous---Coarse-Grained

Ensemble	Grid	Telapsed	Tcomp	Tcomm	MFLOPS	Effm
16	258	244.0	243.7	0.145	49.5	0.078
25	206	143.1	142.9	0.095	81.9	0.082
36	174	105.0	104.9	0.064	116.7	0.081
49	150	82.3	82.1	0.073	153.2	0.078
64	130	63.8	63.7	0.060	188.9	0.074
81	114	51.6	51.5	0.077	216.9	0.067
100	100	52.6	52.5	0.059	241.4	0.060
121	94	45.3	45.2	0.042	254.4	0.053

Table 4 shows that the asynchronous algorithm outperforms the synchronous algorithm by a factor of 1.2 to 1.8 and this ratio is quite significant since the domain decomposition was already quite efficient. The measured ratio of floating point operations to data words transferred via message-passing ranged from 37 for the 16-node ensemble to 10 for the 121-node ensemble. The time spent in the computational part of the asynchronous implementation is less than that of the synchronous scheme. The division of the mesh into several blocks produced an increased cache efficiency similar to that gained by stripmining as discussed in section 2.3.

Figure 15 shows similar decreases in efficiency for both versions of the ADI and since the Table 4 showed negligible communication overhead for the asynchronous algorithm, the reason for similar declines appears to be the reduction in computational performance due to the smaller amount of work for the smaller problems. Partitioning the work into 4 quarters to amortize the overhead exaggerates the reduction effect.

Figure 15

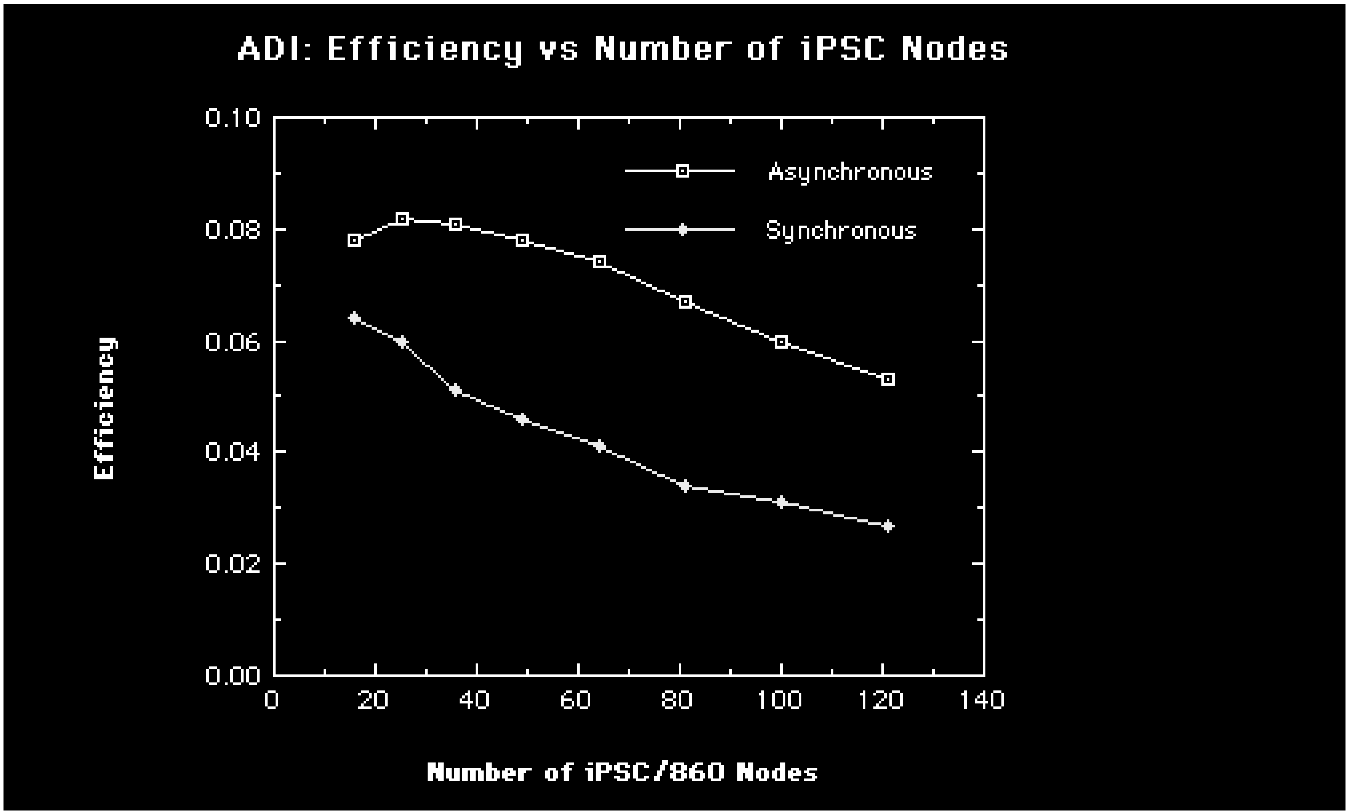


Figure 16 shows the scaling behavior of the ADI ensemble MFLOP rates as a function of the number of iPSC nodes.

Figure 16

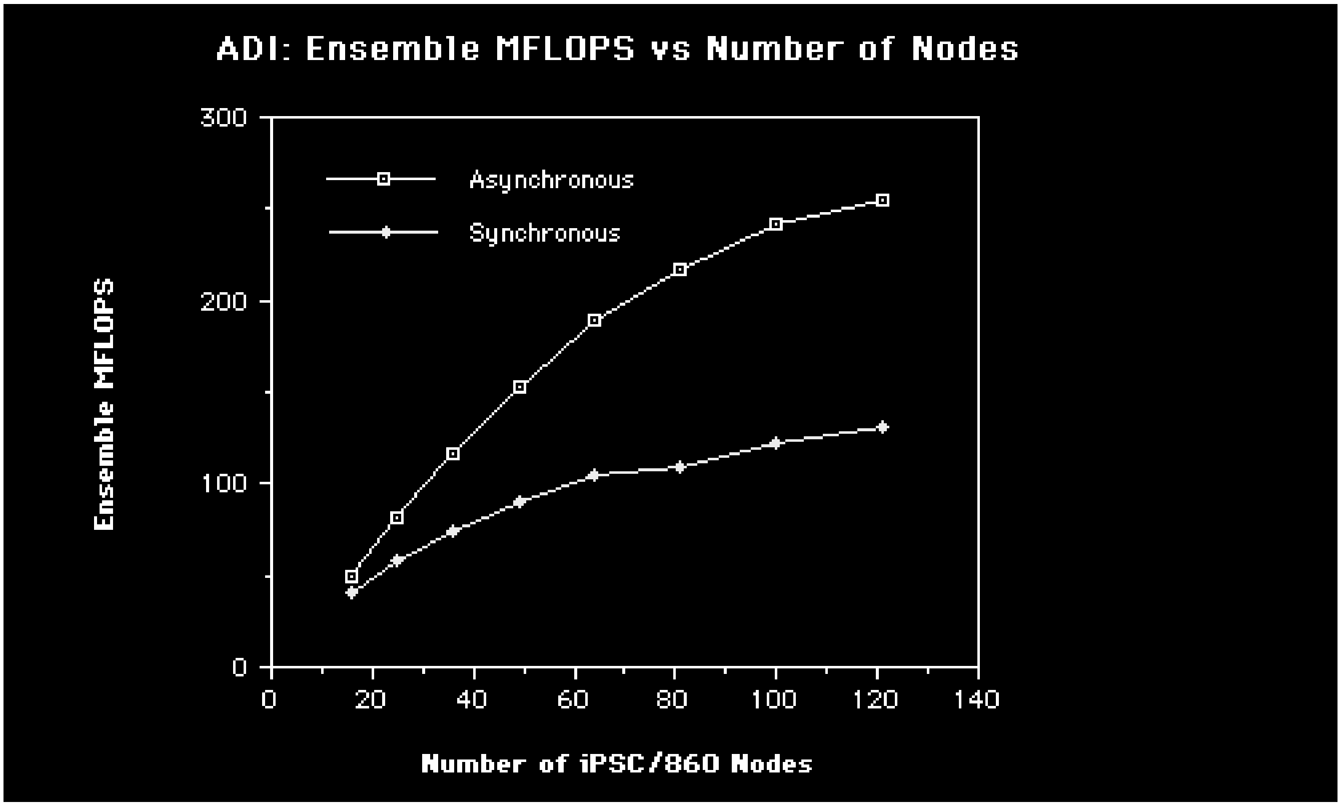


Figure 16 indicates both algorithms display less-than-linear increases in performance. Strong load imbalances and communication delays dominate the synchronous version. Pipelining the algorithm raises the level of node utilization throughout the calculation.

6.4 Lessons for Tools

Implementation of the ADI on the hypercube required considerable modification of the Y-MP code. The extensive communication load associated with the cyclic reduction solver required a different solution algorithm. The smaller amount of data transmission associated with the partitioned solver influenced its selection as a highly parallel tridiagonal solver. Pipelining improved hypercube performance by a factor of 2. The improved ADI algorithm employed three different communication patterns: a one-way transfer to the node immediately "beneath" it in the decomposition, a transfer to and from the driver nodes during the solution phase, and a global sum for convergence checking.

While it may be unlikely for a tool to provide custom solvers, a site may be able to assist its users by providing simple examples of effective solution techniques.

PAT Output

Efficient performance did not occur until the algorithm was pipelined, asynchronous and coarse-grained. While PAT indicated load balance problems, the tool was unable to disclose the insufficient amortization of communication. Figures 17 and 18 show PAT output for the synchronous fine-grained and asynchronous coarse-grained cases. These figures are normalized histograms of the elapsed time spent in the five fundamental tasks. For the NAS configuration, the time denoted by FLICK is the time spent by the node in blocking execution as it waits to receive a message. The figures show different levels of node utilization, and the more efficient version in Figure 18 has a higher level of utilization by the calculation tasks.

The spikes in the node utilization histogram correspond to the driver nodes defined in section 6.2. These nodes spend a considerable amount of time in both calculation and communication. Node 0 has the highest utilization because it serves as a driver node for both sweeps. The PAT output does show that the algorithm has a load balancing problem.

However, a static representation does not help the user to understand whether the pipelining is effectively allowing nodes to overlap communication with computation. The user needs a dynamic representation to see if the nodes are busy while communication occurs. Observation of the CPU status lights on the hardware cabinets provided this information and the next generation of the Intel supercomputer will apparently make this information available even to remote users.

Figure 17

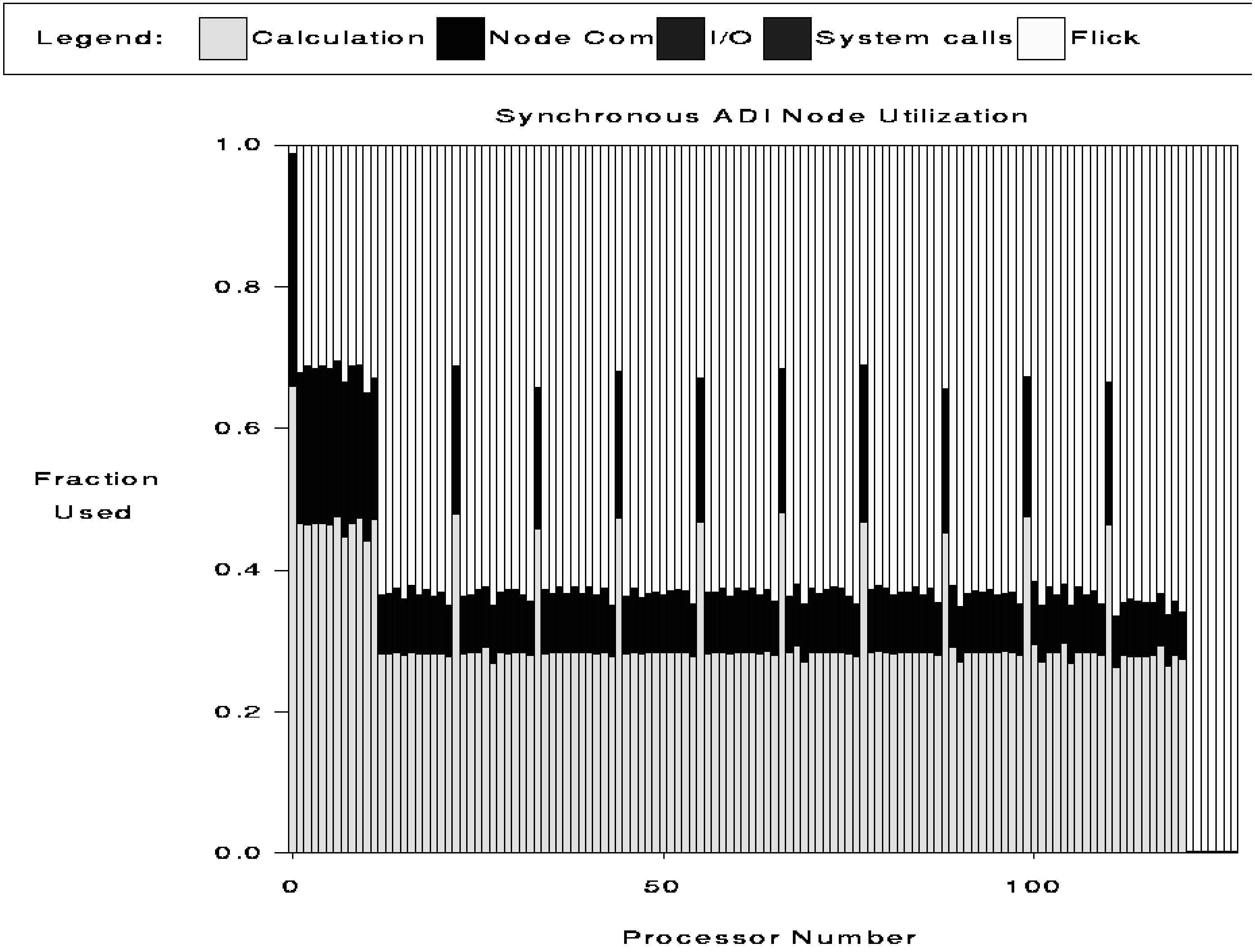
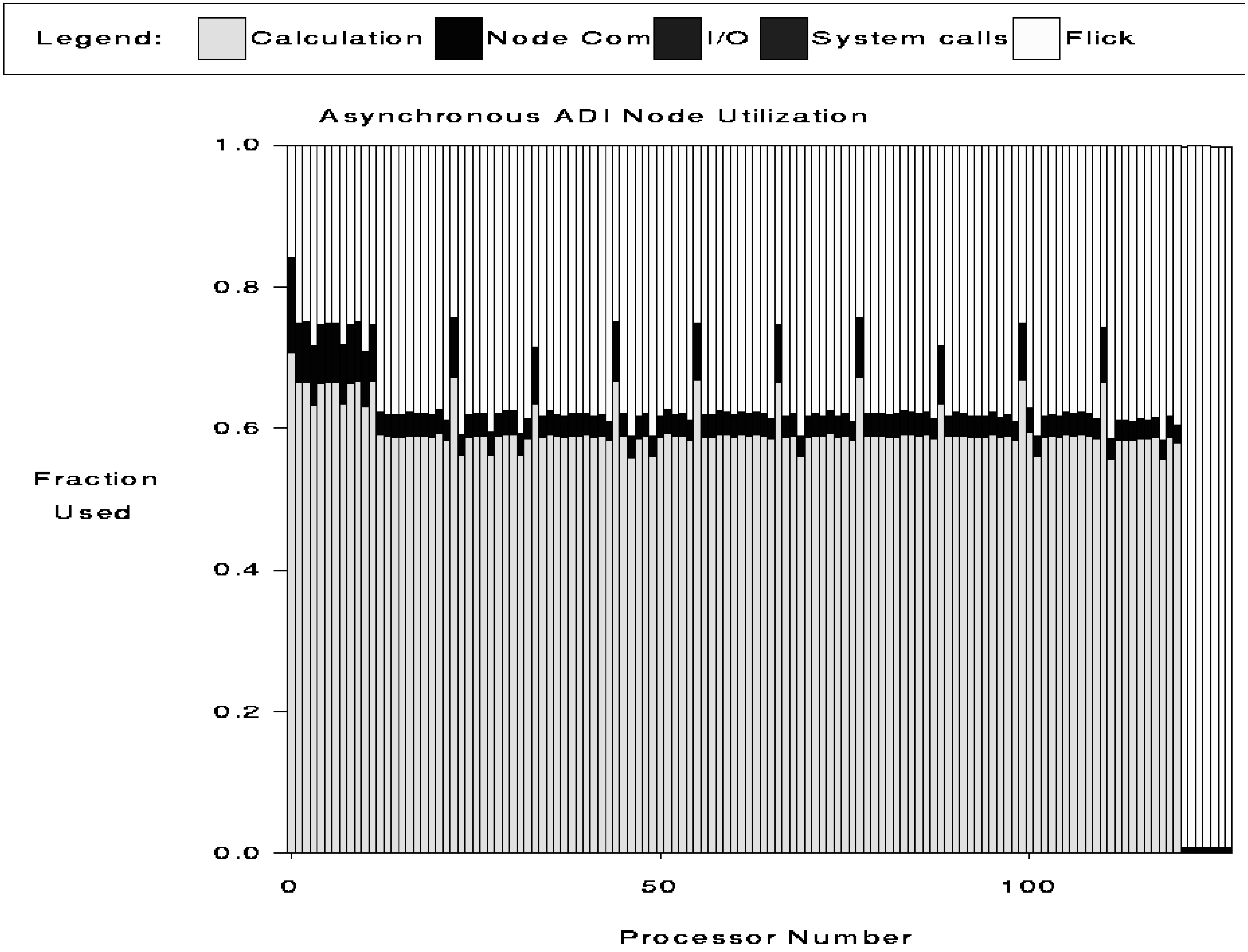

Legend:
Calculation
Node Com
I/O
System calls
Flick
Synchronous ADI Node Utilization
Fraction Used
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
Processor Number

Figure 18



7.0 Multigrid Algorithm

7.1 Code Description

Multigrid solution techniques are an advanced form of relaxation algorithm which allows NAS users to accelerate the convergence of their CFD codes (NASA, 1990). A simple two-grid iteration technique begins by a number of relaxations on a fine grid followed by a projection of the errors to a coarse grid. Relaxations on the coarse grid are then followed by an interpolation back to the fine grid. Three basic operations comprise the multigrid technique: relaxation, projection, and interpolation.

The multigrid algorithm in the Y-MP version of the suite employs a 3-dimensional version of a standard V-algorithm (McCormick, 1987) to solve

Poisson's equation on a power-of-2 cube. The base case employs 5 grids with 2**7 points on the fine grid and 2**3 points on the coarse grid. The recursion present in a serial visit with a Gauss-Seidel solver to all grid nodes dictated a red/black ordering. A relaxation cycle includes separate visits to the red and the black nodes using separate Gauss-Seidel solvers. For each grid in the descending part of the V, the technique contains multiple relaxation cycles and a projection. For each grid in the ascending part of the V-cycle, the technique contains a relaxation on red nodes, multiple relaxation cycles and an interpolation from the coarse black nodes to the fine black nodes. Subroutines corresponding to the three multigrid operators, *i.e.*, relaxation, projection, and interpolation, consist of triple DO-loops visiting either the red or the black nodes.

To parallelize the vector code, the Cray autotasker, FPP, constructed a parallel region by placing each iteration of the outer loop in parallel. The red iteration employs only black points, which are themselves constant during the red iteration. This technique decouples the calculation of the red planes and, for the black iteration, decouples the calculation of the black planes. For each color on a given grid, calculations on each of the planes are performed in parallel with FPP distributing one plane per processor.

7.2 Modifications for Parallel Execution on the Hypercube

The code presented herein represents a straightforward and almost tool-based port of the Y-MP code to the iPSC/860. The vector Y-MP decomposition assigns each plane of a cube to a processor. The results presented in Section 3 indicate that this method incurs a large communication overhead because each node sends a large amount of data relative to the amount of calculation required. A more efficient two-dimensional hypercube decomposition consisted of a domain decomposition involving nonuniform square regions (Briggs, *et al.*, 1990). On each level, the various subgrids created by this decomposition contain different numbers of points and different overlapping boundaries.

A straight Y-MP port provides an example of an approach frequently employed by current parallel tools. This approach consisted of parallelizing the three multigrid operators at the loop-level.

For the shared memory Y-MP, the data structure consists of solution and right-hand side vectors which are stored contiguously in single arrays. As the V-cycle descends into coarser grids, it involves fewer and fewer planes, but level-dependent offsets allow easy access to the various shared memory locations. These offsets also permit easy memory access on the ascent part of the V-cycle.

For the distributed memory hypercube, each active node has 3 planes in memory, but it requires data from two other nodes. As with the Y-MP version, the fine grid relaxation can exchange data with neighboring locations. In this case, the neighboring locations are in contiguously numbered nodes. Data transfer in the coarser grids required careful routing to duplicate the Y-MP memory activity and the eventual construction of a table listing the Y-MP planes active during each phase of the iteration greatly assisted in the port. Eventually, experience with the data transfers allowed algorithmic expression of all sending and receiving nodes during each level of the multigrid scheme.

7.3 Code Performance

The base case treats a cube with 129 points per edge, which corresponds to a problem size of about 1 million points. Table 5 provides performance data for this port.

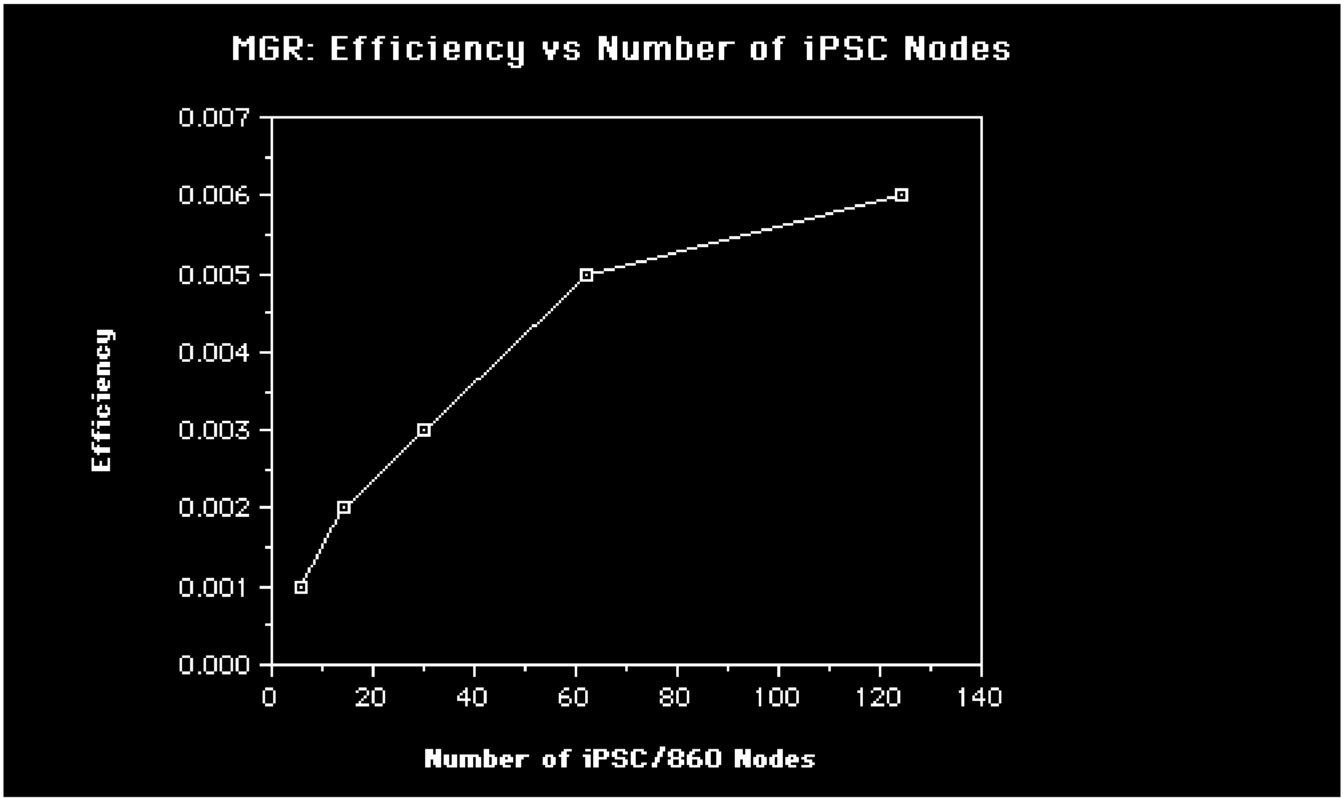
Table 5
MGR iPSC/860 Performance on 3D Problem

Ensem	Telaps	Tcomp	Tcom	MFLO	Effm
ble	ed		m	PS	
6	2.1	1.1	1.0	0.3	0.001
14	4.2	2.1	2.1	1.5	0.002
30	10.6	5.6	5.0	5.4	0.004
62	36.2	19.1	17.1	13.2	0.005
126	131.3	69.7	63.8	30.3	0.006

Table 5 indicates almost equal fractions of time spent in computation and message-passing. The measured ratio of floating point operations to data words sent via message-passing ranged from 0.5 for the 6-node problem to 0.45 for the 126-node problem. This low ratio is the reason for the poor performance displayed by this decomposition. The small increase in node efficiency is apparently due increased cache efficiency.

Figure 19 shows the efficiency for this decomposition.

Figure 19



7.4 Lessons for Tools

The port of the MGR to the iPSC/860 attempted to maintain the Y-MP problem and performed poorly because of heavy communication loads. This result is reminiscent of the initial port of the SOR to the Y-MP described in Section 3. A more effective port may require a nonuniform decomposition which would be very difficult to implement with a parallel tool.

8.0 Discussion

The previous sections have illustrated the types of changes required to convert various Y-MP vector codes into iPSC/860 codes. The Cray autotasker, FPP, produced shared memory parallel versions of the vector codes, generally requiring only minor algorithm modification to improve the parallel efficiency. The original vector codes treated problems with simple geometries and FPP produced highly regular, load-balanced, parallel decompositions.

The porting of these vector codes to the distributed memory iPSC/860 provides examples of reasonably efficient parallel code created from vector source code and can assist in the evaluation of parallel tool software. Typical modifications were made at a task level, the next level above that of the DO-loop and an efficient parallel tool would presumably create or assist the user in creating a similar source. Utilization of the manual ports as a template could allow constructive criticism of source files created by inefficient parallel tools.

The following section reviews the changes required to port the vector codes to the iPSC/860 in terms of the actions required by the parallel tools. Section 8.2 briefly discusses the capabilities of the more prominent tools and compares the current port with some previous efforts.

8.1 Summary of the iPSC/860 Changes

NAS users typically employ vectorized code in programs involving explicit solvers, relaxation algorithms, and implicit solvers. The following discussion reflects lessons learned from the parallel suite with these three general algorithms.

Explicit Solvers

The SWM code used a standard explicit algorithm in which the new values at each node depend only on the old values and those of the nearest neighbors. The iPSC/860 porting effort combined a simple domain decomposition with a nearest neighbor communication scheme to create a distributed memory parallel code. Satisfaction of the periodic boundary conditions required special coding to identify the nodes transmitting and receiving the remote neighbor messages.

For the SWM, the parallel tool would have to generate both a task-level decomposition and also a schedule for the periodic boundary conditions.

Relaxation Algorithms

The vectorized iterative solvers, SOR and MGR, employed red/black schemes, implemented in the case of the SOR with indirect addressing, and implemented in the case of MGR with indexing offsets.

A straightforward port of the SOR which employed the Y-MP domain decomposition produced poor performance because this decomposition led to high communication loads relative to the computational work. Replacement of this decomposition with one containing a reduced communication load gave a factor of 20 speedup in performance.

As with the SOR, a straightforward port of the MGR produced poor performance because the Y-MP data decomposition led to high communication loads relative to the computational work. Replacement of this decomposition with one that better suited the iPSC/860 architecture was not possible during this time frame, but examination of other distributed memory multigrid ports indicated that a specially tailored decomposition was required to obtain efficient hypercube performance.

For the SOR, the parallel tool would have to generate both a task-level domain decomposition and a nearest-neighbor communication schedule. To port the Y-MP version of the MGR, the parallel tool must generate a domain decomposition which is custom-made and nonuniform. The iPSC/860 version presented here decomposed the problem along loop-levels and performed very poorly.

Implicit Solvers and Algorithms

Implicit schemes employ global linear system solvers to obtain the solution of the system at a given timestep, and efficient parallel performance requires both a highly parallel solver and highly parallel pre-solver routines to assemble the left-hand and right-hand sides.

The PARACR code, exemplifying a shared memory global solver, employed a modification to the standard odd-even cyclic reduction to provide an algorithm with high potential parallelism. The heavy communication load imposed by this algorithm forced its replacement by a less communication-intensive, partitioned tridiagonal solver.

The ADI code combined a simple domain decomposition with a standard solution algorithm which employed the partitioned solver for the tridiagonal equations. The initial version did not perform well and was rewritten as a pipelined algorithm. This version performed better, but a factor of 2 improvement was achieved by modifying it further to effectively amortize the communication overhead.

For the PARACR, the parallel tool cannot generate an efficient solver from the memory-intensive vector algorithm. A tool processing such a source could at least notify the user by recording the amount of data transmitted. For the ADI, the parallel tool would have to generate both a task-level mapping and also a loop body pipelining to allow parallel execution of the assembly tasks.

8.2 Previous Work

The development of parallel tools for distributed memory machines has recognized the importance of data decomposition. Effective data distribution is a prime goal of the Fortran D language (Hiranandani, Kennedy, and Tseng, 1991). Data decomposition and loop pipelining are two of the general methods used to test a mapping compiler (Sussman, 1993). Optimizations aimed at allowing a distributed memory machine to efficiently compute inner loops over globally defined data structures have also been proposed (Saltz, *et al.*, 1990). These tools apply certain architecture-specific optimizations to constructs around or within the loops and seem to employ an approach resembling the Cray autotasker, FPP.

None of the above tools attempts to assist the user in a coarse-grained decomposition similar to the ones employed in porting the vector codes to the iPSC/860. Instead, these parallel tools employ various forms of dependency analysis to partition loops by columns or rows. While loop-level code transformations can improve the iPSC/860 cache efficiency, this approach will produce code with high communication loads because it must employ the decomposition specified by the user in the loop itself.

A high-level approach to creating distributed memory code generally involves an abstract approach because it must contain details for creating architecture-specific versions of a user's source code. The MUPPET utility provides an abstract machine layer and a topological mechanism for efficient mapping of the abstract machine onto the multiprocessor (Muhlenbein, *et al.*, 1988). The PIE approach provides a set of basic parallel algorithms, termed implementation machines, and supplies the necessary control, communication, synchronization, and activity decomposition (Segall and Rudolph, 1985). High-level approaches may have suffered in the past by trying to incorporate many different parallel architectures and task decompositions.

Since the successful iPSC/860 ports all employed high-level decompositions, these high-level approaches would seem to deserve additional support and further investigation. Other investigators (Singh and Hennessy, 1992) have also noted the primary role played by domain decompositions in constructing efficient distributed memory programs. These results seem to argue for a high-level tool with specific capabilities.

8.3 Observations for Preprocessors

A data decomposition tool producing efficient hypercube performance will provide effective data transmittal. The manually-obtained iPSC/860 versions of the vector Y-MP codes presented herein imply that data decomposition will be an ineffective approach to creating efficient parallel code on distributed memory multiprocessors. However, any decomposition provided by a parallel tool should measure the amount of message-passing which occurs during the problem execution. Users can measure the effectiveness of decompositions by examining the ratio of the number of floating point operations to the number of data words transmitted. This type of measurement does not require extensive tracking of elapsed time.

8.4 Observations for Postprocessors

The iPSC/860 version of the ADI performed poorly because of inadequate amortization of communication overhead. While the PAT tool did inform the user of load-balancing problems, its static representation did not assist in this problem. Real-time visual observation of the node status lights on the machine cabinet indicated a large amount of processor hold time and motivated the more successful coarse-grain decomposition. A tool should be able to access such signals and transmit the information to the user.

Distributed memory tools should provide guidance to the user in a manner similar to the *atexpert* tool (Cray, 1990).

Execution of the fine-grain and coarse-grain ADI code with the AIMS postprocessor (Fineman, 1992) would be a worthwhile exercise to see if this tool can provide information equivalent to the visual observation of the iPSC/860 node status lights.

9.0 Conclusions

The results of porting several vector codes to the iPSC/860 indicate that effective use of a distributed memory multiprocessor requires code design features not present in vector code. These include an architecture-dependent data distribution and the pipelined execution of algorithms.

A high-level domain decomposition created four successful iPSC/860 codes whereas the loop-level decomposition created a poorly performing iPSC/860 code. Domain decompositions are effective on this architecture because they possess strong data locality and load-balancing characteristics. Such decompositions also display computation burdens proportional to the volume enclosed by the domain and data communication requirements defined by the domain perimeter. This property leads to excellent scaling characteristics for algorithms employing domain decompositions.

Mapping the data in a manner which exploits the multiprocessor architecture is a necessary but not always sufficient condition for effective hypercube performance. Efficient mappings may be the most important factor in porting explicit solvers and relaxation techniques to the hypercube, but efficient mappings alone will not lead to high performance implicit solvers. Processors tend to idle during the global data transfers involved with implicit solvers, and codes must overlap communication with computation to achieve high node utilization. Pipelining is one fairly simple technique to achieve such overlap.

The implementation issues associated with parallel programming, communication and synchronization, were easy to resolve after the high-level decomposition. The iPSC parallel programs employed standard Fortran constructs supplemented with a few communication calls and some global arithmetic functions.

The usefulness of domain decomposition and pipelining apply quite specifically to the numerical problems solved by NAS users, time-dependent, partial differential equations. These conclusions strongly support the adoption or development of a specific type of high-level parallel tool, one with the domain decomposition and pipelining templates. Users of such a tool must understand the algorithms used by their code, but the results presented in this report indicate a strong possibility of an effective parallel tool.

While the Intel debugger IPD helped to resolve some difficult message-passing problems, resolution of performance issues in asynchronous algorithms could benefit from a dynamic representation of the Intel performance tool (PAT) data.

10.0 Acknowledgements

Thanks to Dave Dinucci and Russell Carter for reviewing this paper.

11.0 References

Adams, L. F. and Jordan, H. F. 1986. “Is SOR Color-Blind?”, SIAM J. Sci.

Stat. Comput., 7, 2, 490-506.

Bailey, D. H., et al. 1991. "The NAS Parallel Benchmarks-Summaries and Preliminary Results" in *Proceedings of the Supercomputing '91 Conference*, pp. 158-165.

Bailey, D. H. 1993. "RISC Microprocessors and Scientific Computing", RNR Technical Report RNR 93-004, Ames Research Center, Moffett Field CA, 1993.

Bergeron, R. J. 1993. "Automatic Parallelization Tools Efficiency: FPP/ Atexpert Case Studies", NAS Technical Report RND 93-011, Ames Research Center, Moffett Field CA.

Briggs, W., Hart, L, McCormick, S.F., and Quinlan, D. 1988. "Multigrid Methods on a Hypercube" in *Multigrid Methods: Theory, Applications, and Supercomputing*, S.F. McCormick, editor, Lecture Notes in Pure and Appl. Math., Marcel Dekker, New York, pp. 63-83.

Case, B. 1986. "Intel i860 Performance" in *A Guide to RISC Multiprocessors*, M. Slater, editor, Academic Press, New York, NY.

Carr, S., and Kennedy, K. 1992. "Compiler Blockability of Numerical Algorithms" in *Proceedings of the Supercomputing '92 Conference*, pp. 114-124.

Chen, D., and Pase, D. 1991. "An Evaluation of Automatic and Interactive Parallel Programming Tools", in *Proceedings of the Supercomputing '91 Conference*, pp. 412-423.

Cray Research Inc. 1990. *UNICOS Performance Utilities Reference Manual*, Pub. No. SR-2040 6.0, Cray Research Inc.,1990.

Fineman, C., Hontalas, P., Listgarten, S., and Yan, J. 1992. "A User's Guide to AIMS-The Ames Instrumentation System, Version 1.3." NASA Ames Research Center, Moffett Field CA.

Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., Walker, D. W. 1990. *Solving Problems on Concurrent Processors*,
Prentice-Hall, Englewood Cliffs, New Jersey 07632.

Hiranandani, S., Kennedy, K., and Tseng, C. 1991."Compiler Optimizations for Fortran-D on MIMD Distributed-Memory Machines", in *Proceedings of the Supercomputing '91 Conference*, pp. 86-100.

Hockney, R. and Jesshope, C. 1988. *Parallel Computers 2*.,Bristol, England: Adam Hilger Ltd.

Hoffman, G.R., Schwartztrauber, P.N., and Sweet, R.A. 1986. "Aspects of Using Multiprocessors in Meteorological Modelling" in *Multiprocessing* *in Meteorological Models*, G.R. Hoffman and D.F. Snelling, eds, Springer-Verlag, New York, NY.

Intel Corporation. 1992. “iPSC/860 Parallel Performance Analysis Tools Manual” Intel Supercomputer Systems Division, Beaverton Oregon.

Johnson, S. L., Saad, Y., and Schultz, M.H. 1987. “Alternating Direction Methods on Multiprocessors”, SIAM J. Sci. Stat. Comput., 8, 7, 686-700.

Karp, A. and Flatt, H. P. 1989. “Measuring Parallel Processor Performance”, Communications of the ACM,33,5 (May 1990) 539-543.

Lee, K. 1991. “Achieving High Performance on the iPSC/860”, Technical Report RNR 91-029, October, 1991, Ames Research Center, Moffett Field CA 94035.

Mavriplis, D. J., Das, R., Saltz, J. and Vermeland, R. E. 1992. “Implementation of a Parallel Unstructured Euler Solver on Shared and Distributed Memory Architectures” in *Proceedings of the Supercomputing ‘92 Conference*, pp. 132-141.

McCormick S.F.,editor, 1987. *Multigrid Methods*, SIAM, Philadelphia, Pa.

Muhlenbein, H. Kramer, O., Limburger, F., Mevenkamp, M., and Streitz, S. 1988. “MUPPET: A programming environment for message-based multiprocessors”, Parallel Computing, 8, 201-221.

National Aerodynamics and Space Administration (NASA), 1990. *NAS Technical Summaries*, Ames Research Center, Moffett Field, California 94035-1000.

Nitzberg, B. 1992. “Performance of the iPSC/860 Concurrent File System”, NAS Technical Report RND 92-020, Ames Research Center, Moffett Field CA, 1992.

Press, W. H., B.P. Flannery, S.A. Teulolsky, and W. T. Vetterling 1986. *Numerical Recipes*, Cambridge University Press, Cambridge.

Sadourny, R. 1975. “The Dynamics of Finite-Difference Models of the Shallow Water Equations”, J. Atmos. Sci., 32, 680-689.

Saltz, J., Crowley, K., Mirchanandey, R., and Berryman, H. 1990. “Run-Time Scheduling and Execution of Loops on Message Passing Machines”, Journal of Parallel and Distributed Computing, 8, 303-312.

Segall, Z., and Rudolph, L. 1985. “PIE - A Programming and Instrumentation Environment for Parallel Processing”, IEEE Software, Vol 2, November 1985, pp 22-37.

Singh, J. P., and Hennessy,J. L. 1992. “Finding and Exploiting Parallelism in an Ocean Simulation Program”, Journal of Parallel and Distributed Computing, 15, 27-48.

Stone H.S., 1987. *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, Menlo Park, California.

Wang, H. H. 1981. “A Parallel for Tridiagonal Equations”, ACM Transactions on Mathematical Software, Vol. 7, No. 2, June 1981, pp 170-183.